Transbase®
SQL Reference Manual

Transaction Software GmbH
Willy-Brandt-Allee 2
D-81829 München
Germany
Phone: +49-89-62709-0
Fax: +49-89-62709-11
Email: info@transaction.de
http://www.transaction.de

Version 6.8.1.40
November 02, 2010

Contents

Chapter 1

Introduction

TB/SQL is the data retrieval, manipulation and definition language for the relational data base system Transbase. TB/SQL is an SQL implementation compatible to the ISO/DIS standard 9075. TB/SQL realizes most of SQL2 (Intermediate LEVEL) and additionally functional extensions which make the SQL language more powerful and easier to use.

This manual is intended for users who already have a basic knowledge of SQL. Heavy use of examples is made in order to clarify syntactic or semantic questions. All examples refer to a sample database outlined in Appendix A of this manual.

Chapter 2

General Concepts

2.1 Conventions for Syntax Notation

Brackets `[ ]` are delimiters for an optional part.

The vertical line `|` separates alternatives.

Braces `{ }` group several items together, e.g. to form complex alternatives. They are functionally equivalent to the standard braces `()` as used in arithmetic expressions. An ellipsis `...` indicates that the preceding item may be repeated arbitrarily often. To distinguish terminal from non-terminal symbols, all non-terminal symbols start with an uppercase letter followed by lowercase letters, all terminal symbols are represented by themselves. All keywords are written in uppercase letters.

2.2 DataType

A *DataType* specifies the type of a field or the target type for a type conversion.

Syntax:

```
DataType ::=
    TINYINT
  | SMALLINT
  | INTEGER
  | BIGINT
  | NUMERIC [(Precision [,Scale])]
  | DECIMAL [(Precision [,Scale])]
  | FLOAT
```

```
  | DOUBLE
  | REAL
  | VARCHAR [(Precision)]
  | CHAR [(Precision)]
  | CHAR(*)
  | STRING
  | BINCHAR [(Precision)]
  | BINCHAR (*)
  | BITS (Precision)
  | BITS (*)
  | BITS2 (Precision)
  | BITS2 (*)
  | BOOL
  | DATETIME Range
  | DATE
  | TIME
  | TIMESTAMP
  | TIMESPAN Range
  | INTERVAL StartSpec [ TO EndSpec ]
  | BLOB

Precision ::=
    IntegerLiteral

Scale ::=
    IntegerLiteral

Range ::=
    LeftBr RangeIx1 [:RangeIx1] RightBr

LeftBr ::=
    [

RightBr ::=
    ]

RangeIx1 ::=
    YY | MO | DD | HH | MI | SS | MS

StartSpec,
EndSpec    ::= RangeIx2 [Precision]

RangeIx2    ::=
    YEAR | MONTH | DAY | HOUR | MINUTE | SECOND
```

Explanation: Each field of a table has a data type which is defined at creation time of the table. Constant values (Literals) also have a data type which is derived by the syntax and the value of the Literal.

TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, FLOAT, DOUBLE are called the arithmetic types. DECIMAL is a synonym for NUMERIC.

Precision in NUMERIC is the maximum total number of digits. Scale is the number of digits behind the decimal point.

If Scale in NUMERIC is omitted, it is equivalent to 0. If Precision in NUMERIC is omitted it is equivalent to 30.

CHAR, VARCHAR(p), CHAR(p), CHAR(*), BINCHAR, BINCHAR(p), BINCHAR(*), BITS(p), BITS(*), BITS2(p), BITS2(*) are called the character types.

CHAR is equivalent to CHAR(1). Values of type VARCHAR(p) are variable length character sequences of maximum length p bytes. Values of type CHAR(p) are fixed sized character sequences of length p bytes. Values of type CHAR(*) are variable length character sequences up to a maximum length of 4000 Bytes unless the database page size restricts tuples in length further (see also the chapter "Transbase Restrictions and Resource Limitations" of the system guide). No character can be the binary 0 character. In their internal representation, VARCHAR and CHAR values are ended with a binary 0 character.

BINCHAR is equivalent to BINCHAR(1). Values of type BINCHAR(p) are fixed sized character sequences of length p bytes. Values of type BINCHAR(*) are variable length character sequences. Characters in BINCHAR also can be the binary 0 character. In their internal representation, BINCHAR values have a length field.

Values of type BITS(p), BITS2(p), BITS(*), BITS2(*) are fixed sized or variable sized bit sequences, resp. Their internal representation resembles BINCHAR whose length is rounded up to the next multiple of 8 bits. BITS2 is more space economic than BITS because it uses a 2-byte length field in contrast to 4-byte in BITS. The maximum value of p (number of bits) is 31968 for both variants. BITS possibly will allow a higher range in future versions.

See chapter 10 (The TB/SQL Datatypes BITS(p) and BITS(*)).

Transbase V5.3: Please note that for UTF-8 databases, ASCII characters take exactly one byte, non-ASCII characters may take up to 6 bytes depending on their unicode values as follows:

The ordering of character sequences (strings) is the lexicographic extension over the collating sequence of characters. Characters are ordered according to the underlying machine code (e.g. ASCII). But see also the notes about user defined sortorders in chapter 2.3 (User Defined Sortorders).

Unicode value	Bytes
$2^0 \ldots 2^7 - 1$	1
$2^7 \ldots 2^{12} - 1$	2
$2^{12} \ldots 2^{17} - 1$	3
$2^{17} \ldots 2^{22} - 1$	4
$2^{22} \ldots 2^{27} - 1$	5
$2^{27} \ldots 2^{32} - 1$	6

BOOL is called the logical type. Its values are TRUE and FALSE. The ordering of logical values is: FALSE is less than TRUE.

DATETIME,DATE,TIME,TIMESTAMP and TIMESPAN,INTERVAL are called the time types. They are used to describe points in time or time distances, resp. Their semantics is described in detail in a separate chapter 9

BLOB is the type for binary large objects. The type is described in a separate chapter within this manual.

See table 2.1 for a summary of data types and ranges.

MAXSTRINGSIZE is 4000 Bytes. The maximum length of a character sequence as a field inside a tuple may be less according to the condition that a tuple must always fit into a page. The pagesize is a database specific parameter chosen at creation time (see System Guide).

Whenever character sequences are needed which are longer than the above described limits, then the data type BLOB must be used.

2.3 User Defined Sortorders

For the character types (VARCHAR / CHAR / BINCHAR), the ordering of values is defined by the machine code (by default). However, this ordering may be dynamically set (overruled) at the programming interfaces TBX and ESQL ("user defined sortorder"). A user defined sortorder only affects the evaluation of an explicit ORDER BY Clause for character result fields (compare *SelectStatement* within this Manual). It does not affect the physical layout of tuples within the tables nor the result of string comparison in SearchConditions. The setting of a user defined sortorder is explained in detail in the corresponding manuals for application programming (TBX / ESQL).

2.4 Type Compatibility

Whenever values serve as operands for an operation, their types must be compatible. The compatibility rules are as follows:

- All arithmetic types are compatible among each other.
- All character types are compatible among each other.
- The logical type is compatible with itself.
- The compatibilities of time types among each other and with other types is described in Chapter 5.

Arithmetic data types are ordered according to the following type hierarchy:

DOUBLE	Highest Arithmetic Type
FLOAT	
NUMERIC	
BIGINT	
INTEGER	
SMALLINT	
TINYINT	Lowest Arithmetic Type

Character data types are ordered according to the following type hierarchy:

BITS	Highest Character Type
BINCHAR	
(VAR)CHAR	Lowest Character Type

If values of different arithmetic types are involved in an operation, they are implicitly converted to the highest type among them before the operation is performed. Upward conversion within arithmetic types never causes loss of significant digits, but note that values converted to FLOAT or DOUBLE are not always exactly representable.

If two values of type VARCHAR, CHAR or BINCHAR with different length are compared, then the shorter string is padded with the space character `' '` up to the length of the longer string.

If two character types are processed in operations UNION, INTERSECTION and DIFF then the following rules apply for determinig the result type: One particpating CHAR(*) yields CHAR(*). 2 input types of CHAR or VARCHAR with precisions p1 and p2 yield output precision p = maximum(p1,p2). If at least one of them is VARCHAR - then VARCHAR(p) is the result type else CHAR(p).

For operations on type BITS see chapter 10 (The TB/SQL Datatypes BITS(p) and BITS(*)).

2.5 Type Exceptions and Overflow

A type exception is the event that a value fails to be in the range of a requested type. The following operations may cause a type exception:

1. Arithmetic computation on values (addition, subtraction etc.)
2. Insertion or Update of tuples.
3. Explicit casting of a value to a different type (CAST-operator).
4. Assignment of a field value to a host variable of an Embedded SQL program (see ESQL Manual).

In each of these operations, the requested type is defined as follows.

In case (1) - arithmetic computation - the type of the result value is requested to be the same as that of the input operands.

Example: The expression

```
1000000 * 2
```

is legal, because the input type is INTEGER and the result is still in the range of INTEGER.

The expression

```
1000000 * 1000000
```

leads to a type exception because the result is no more in the range of INTEGER.

To avoid this, it would be sufficent to cast one (or both) input operands to a higher ranged type e.g. NUMERIC(30,0)

```
1000000 CAST NUMERIC(30,0) * 1000000
```

or to write one operand as a NUMERIC constant

```
1000000.0 * 1000000
```

In case (2) - Insertion or Update of tuples - the requested types are those of the corresponding fields of the target table.

Example: With a table T with a field f of type TINYINT, the following statement would cause a type exception:

```
INSERT INTO T (f) VALUES (200)
```

In case (3) - explicit casting - the requested type is the explicitly specified type in the CAST-operator.

Example: The expressions

```
100 CAST SMALLINT         -- legal
'hello' CAST CHAR(10)     -- legal
```

are legal (the latter example pads the string with 5 blanks at the end).

The expressions

```
200 CAST TINYINT          -- illegal
'hello' CAST CHAR(3)      -- illegal
132.64 CAST NUMERIC(4,2)  -- illegal
```

are illegal, since they cannot be converted into the requested types because of overflow.

In case (4) - assignment to a host variable - the requested type is that of the host variable. See the ESQL Manual for the correspondence between TB/SQL types and types of the host languages.

2.6 CASTing Incompatible Types from and to CHAR

As described in chapter 2.4 (Type Compatibility), there are several groups of types, namely arithmetic, character, logical and time types and the BLOB type. All types within one group are compatible among each other.

Additionally, with the exception of type BLOB, there is the possibility to convert values of each type to the type CHAR(*) and (VAR)CHAR(p). This is done by the CAST operator.

2.6.1 CASTing to CHAR

The main usage of casting to CHAR is to make string operations (e.g. the LIKE operator or the string concatenation operator '+') available to other types.

For example, assume that field 'birthday' of type DATETIME(YY:DD) is in table person. The following pattern search is possible:

```
SELECT *
FROM person
WHERE birthday CAST CHAR(*) LIKE '%19%19%19%'
```

As another example, assume that field 'price' of type NUMERIC(6,2) is in table 'article'. The following query extracts all articles with prices ending in .99:

```
SELECT *
FROM article
WHERE price CAST CHAR(*) LIKE '%.99'
```

As a further example, assume that fields 'partno1' and 'partno2' of type INTEGER are in table 'parts'. The following query constructs a composed partno of type CHAR(*) with a '/' in the middle

```
SELECT partno1 CAST CHAR(*) + '/' + partno2 CAST CHAR(*)
FROM parts
WHERE ...
```

2.6.2 Casting from CHAR

Field values of type CHAR(*) or (VAR)CHAR(p) can be casted to any type (except BLOB) provided that the source value holds a valid literal representation of a value of the target type.

For example, assume that in a table t a field f with type CHAR(*) or VARCHAR(p) contains values of the shape xxxx where xxxx is a 4-digit number. After CASTing to INTEGER one can perform arithmetic calculations with the values.

```
SELECT f CAST INTEGER + 1 , ...
FROM t
WHERE ...
```

Note that an error occurs if the source value is not a valid literal of the target type.

2.7 Literal

Literals are constants of a certain type.

```
Literal ::=
    IntegerLiteral
  | NumericLiteral
  | RealLiteral
  | StringLiteral
  | BoolLiteral
  | DatetimeLiteral
  | TimespanLiteral
```

Explanation: All Literals except *DatetimeLiteral* and *TimespanLiteral* are defined in the following paragraphs. *DatetimeLiteral* and *TimespanLiteral* are defined in Chapter 5.

2.7.1 IntegerLiteral

An *IntegerLiteral* is the representation of a constant number without fractional part.

An *IntegerLiteral* is a non-empty sequence of digits. Note that, by definition, an *IntegerLiteral* is a positive number without a sign. A negative number is obtained by applying the unary minus operator to an *IntegerLiteral* (see section 5.3 on *Expression*s below). Therefore, a separator is permitted between an eventual unary minus and an *IntegerLiteral*, whereas no separators are permitted within the sequence of digits.

Each *IntegerLiteral* has a data type which is either INTEGER, BIGINT or NUMERIC with scale 0. The data type is derived by the value of the *IntegerLiteral*: if the value is inside the range of INTEGER then the type is INTEGER. If the INTEGER range is not sufficient and the value is inside the range of BIGINT then the type is BIGINT else the type is NUMERIC(p,0) where p is the number of digits of the literal.

Example:

```
5                       -- INTEGER
33000                   -- INTEGER
-33000                  -- INTEGER
1234567890123           -- BIGINT
12345678901234567890123 -- NUMERIC(23,0)
```

2.7.2 NumericLiteral

A *NumericLiteral* is the representation of a constant number with fractional part.

A *NumericLiteral* either is an *IntegerLiteral* followed by a decimal point or is an *IntegerLiteral* followed by a decimal point and another *IntegerLiteral* or is an *IntegerLiteral* preceded by a decimal point. *NumericLiteral* again is a positive number, by definition.

The data type of a *NumericLiteral* is NUMERIC(p,s) where p is the total number of digits (without leading 0's in the non fractional part) and s is the number of digits behind the decimal point.

Example:

```
13.       -- NUMERIC(2,0)
56.013    -- NUMERIC(5,3)
 0.001    -- NUMERIC(3,3)
  .001    -- NUMERIC(3,3)
```

The last two representations are equivalent.

2.7.3 RealLiteral

A *RealLiteral* is the representation of a constant number with mantissa and exponent.

A *RealLiteral* is an *IntegerLiteral* or a *NumericLiteral*, followed by 'e' or 'E', followed by an optional minus or plus sign followed by another *IntegerLiteral*. A *RealLiteral* again is a positive number, by definition.

Each *RealLiteral* has a data type which is FLOAT or DOUBLE. The data type is derived by the value of the *RealLiteral* (see table 2.1).

Example:

```
5.13e10     -- FLOAT
5.13e+10    -- FLOAT
0.31415e1   -- FLOAT
314.15E-2   -- FLOAT
314e-2      -- FLOAT
-  314e-2   -- FLOAT
1.2e52      -- DOUBLE
```

Note that no separators are allowed within *RealLiteral*, but are allowed between an eventual unary minus and a *RealLiteral*. The next example shows incorrect *RealLiteral*s:

Example:

```
3.14 e4   -- illegal
3.98E -4  -- illegal
3.98e- 4  -- illegal
```

2.7.4 StringLiteral, CharLiteral, BincharLiteral, BitsLiteral

A *StringLiteral* is the representation of a constant string.

Syntax:

```
StringLiteral ::=
    CharLiteral
  | BincharLiteral
  | BitsLiteral

CharLiteral ::=
    QuotedStringLiteral | UnicodeLiteral

QuotedStringLiteral ::=
    sequence of characters enclosed in single quotes

UnicodeLiteral ::=
    0u followed by sequence of hexadecimal characters

BincharLiteral ::=
    0x followed by sequence of hexadecimal characters

BitsLiteral ::=
    0b followed by sequence of 0 and 1,
    see also Chapter ''The TB/SQL Datatypes BITS(p) and BITS(*)''
```

Explanation: A *QuotedStringLiteral* is a (possibly empty) sequence of characters in single quotes. If a single quote is needed as character, it must be written twice, as shown in the examples. The data type of a *CharLiteral* is CHAR(p) where p is the number of characters without the surrounding quotes.

A *UnicodeLiteral* is 0u followed by a number of hexadecimal characters, four per each Unicode character. The data type of a *UnicodeLiteral* is CHAR(p) where p is the size of the UTF-8 coded equivalent.

A *BincharLiteral* is 0x followed by a (possibly empty) even number of 0, 1..9, a..f, A..F. The data type of a *BincharLiteral* is BINCHAR(p) where p*2 is the number of hexadecimal characters.

Each *CharLiteral* can be expressed by a *BincharLiteral* but not vice versa. Of course, if one expresses printable characters as a *BincharLiteral*, the semantics becomes machine code dependent.

Example:

```
'xyz'                                    -- CHAR(3)
'string with a single quote '' inside'   -- CHAR(35)
'single quote '''                        -- CHAR(14)
''                                       -- CHAR(0)
0u006D00fc006E                           -- CHAR(4)
0ufeff                                   -- CHAR(3)
0xA0B1C2                                 -- BINCHAR(3)
0xA0B                                    -- illegal
0uFC                                     -- illegal
0u00FC                                   -- CHAR(2)
```

2.7.5 BoolLiteral

A *BoolLiteral* is the representation of a constant boolean value.

Syntax:

```
BoolLiteral ::=
    FALSE | TRUE
    Boolean values are ordered: FALSE is less than TRUE.
```

2.7.6 Identifier

Syntax:

```
Identifier::=
    StandardIdentifier
  | DelimiterIdentifier

StandardIdentifier ::=
    LetterDigit ...
```

```
LetterDigit ::=
    <Letters a-z, A-Z, digits 0-9, underscore>

DelimiterIdentifier ::=
    "Character ..."

Character ::=
    <each printable character except double quote>
```

Explanation: A *StandardIdentifier* is a sequence of letters (`a-z`, `A-Z`, `_`) and digits (`0-9`), where the first character is a letter.

Identifiers are treated case sensitive if the database has been created as case sensitive (and not switched to case insensitive).

Identifiers are treated case insensitive if the database has been created as case insensitive or has been switched to case insensitive.

Keywords (see chapter 2.7.8 below) cannot be used as identifiers. The maximum length of *StandardIdentifier*s is 30.

A *DelimiterIdentifier* is any sequence of printable characters (except the double quote ") surrounded by double quotes. Uppercase and lowercase letters are treated as different letters. Also keywords can be used as *DelimiterIdentifier*s. The maximum length of *DelimiterIdentifier*s is 30 (not including the delimiting double quotes).

Example: (valid identifiers)

```
suppliers
Suppno
xyz_abc
q1
q23p
"5xy"
"select"
```

Example: (invalid identifiers)

```
5xy
select
SeLecT
x:y
?x
"as"df"
```

2.7.7 User Defined Names

Throughout this manual the following rules are used:

Syntax:

```
TableName ::=
    Identifier

ViewName ::=
    Identifier

IndexName ::=
    Identifier

TriggerName ::=
    Identifier

FieldName ::=
    Identifier

CorrelationName ::=
    Identifier

UserName ::=
    Identifier
```

2.7.8 Keywords

TB/SQL keywords are listed in Appendix A. Keywords cannot be used as identifiers.

Note: Keywords are case-insensitive. All keywords shown in the following example are valid.

Example:

```
SELECT from wHerE
```

2.7.9 Separator

In a TB/SQL-statement, keywords, identifiers and literals must be separated from each other by at least one separator. As in many programming languages, possible separators in TB/SQL are the space character (blank), the tab character and the newline character.

In all other places separators are permitted but not needed.

Datatype	Description	Range
TINYINT	1-byte integer	absolute value less than 128
SMALLINT	2-byte integer	absolute value less than 32768
INTEGER	4-byte integer	absolute value less than 2147483648
BIGINT	8-byte integer	absolute value less than 9223372036854775808
NUMERIC(p,s)	exact numeric	precision p ($1 <= p <= 30$) scale s ($0 <= s < p$)
FLOAT	4-byte floating point	absolute value between 3.0e-39 and 1.0e+38
DOUBLE	8-byte floating point	absolute value between 1.0e-307 and 1.0e+307
VARCHAR(p)	character sequence of maximum length p	p between 1 and MAXSTRINGSIZE
(BIN)CHAR(p)	character sequence of fixed length p	p between 1 and MAXSTRINGSIZE
(BIN)CHAR(*)	variable length character sequence	length between 0 and MAXSTRINGSIZE
BITS(p)	bits sequence of fixed length p	p between 1 and 8 * MAXSTRINGSIZE
BITS(*)	variable length bits sequence	length between 1 and 8 * MAXSTRINGSIZE
BOOL	truth value	TRUE, FALSE
DATETIME	point in time	see Chapter 9.1
DATE	point in time	see Chapter 9.1
TIME	point in time	see Chapter 9.1
TIMESTAMP	point in time	see Chapter 9.1
TIMESPAN	time distance	see Chapter 9.2
INTERVAL	time distance	see Chapter 9.2
BLOB	binary large object	see Chapter 11

Table 2.1: Transbase Datatypes and Ranges

Chapter 3

Data Definition Language

The Data Definition Language (DDL) portion of TB/SQL serves to create or delete tables, views and indexes, to grant or revoke user privileges and to install users and passwords. The DDL consists of a collection of *DataDefinitionStatement*s. Most of them implicitly changes one or several tables of the TB-Catalog.

3.1 Overview of DataDefinitionStatement

Syntax:

```
DataDefinitionStatement ::=
    CreateDomainStatement
  | AlterDomainStatement
  | DropDomainStatement
  | CreateTableStatement
  | AlterTableStatement
  | DropTableStatement
  | CreateViewStatement
  | DropViewStatement
  | CreateIndexStatement
  | DropIndexStatement
  | CreateTriggerStatement
  | DropTriggerStatement
  | CreateSequenceStatement
  | DropSequenceStatement
  | GrantUserclassStatement
  | RevokeUserclassStatement
  | GrantPrivilegeStatement
  | RevokePrivilegeStatement
  | AlterPasswordStatement
```

3.2 CreateDomainStatement

Serves to create a domain in the database. A domain is a named type, optionally with default value and integrity constraints.

Syntax:

```
CreateDomainStatement ::=
   CREATE DOMAIN DomainName
   [ AS ] DataType
   [ DEFAULT Expression ]
   [ DomainConstraint ] ...

DomainConstraint ::=
   [ CONSTRAINT ConstraintName ]
   CHECK (SearchCondition)
```

Explanation: The *CreateDomainStatement* creates a domain with the given *DomainName* and with the specified base data type.

The created domain can be used in *CreateTableStatement*s (for type specifications of fields) and as target type in CAST expressions.

If a DEFAULT specification is given then the created domain has the specified value as its default value else the domain has no default value. See the CreateTableStatement for the default mechanism of fields.

The default expression must not contain any subqueries or field references.

If *DomainConstraint*s are specified then all values of the specified domains are subject to the specified search conditions, e.g. if a tuple is inserted or updated in a table and a field of the table is defined on a domain, then the field value is checked against all domain constraints specified on the domain.

The search condition in *DomainConstraint* must not contain any subqueries or field references. The keyword VALUE is used to describe domain values in the search conditions.

For the check to be performed, the formal variable VALUE in the search condition is consistently replaced by the field value. The integrity condition is violated, if and only if the expression `NOT (SearchCondition)` evaluates to TRUE. See below for the consequences concerning NULLs.

Whenever a domain constraint is violated, Transbase issues an error message which contains the *ConstraintName* if it is specified else an internally generated name. For the sake of clarity, it is therefore recommended to specify explicit constraint names.

The current user becomes owner of the domain.

Privileges: The user must have userclass DBA or RESOURCE. For the definition of userclasses see the chapter *GrantUserclassStatement*.

Catalog Tables: For each domain, at least one entry into the table "sysdomain" is made. This entry also contains a *DomainConstraint* if specified. For each further specified *DomainConstraint*, one additional entry is made.

Example:

```
CREATE DOMAIN Suppno AS INTEGER DEFAULT 0

CREATE DOMAIN Salary  AS NUMERIC(9,2) DEFAULT 0
    CONSTRAINT Salcheck CHECK (VALUE BETWEEN 60000 AND 150000)

CREATE DOMAIN MyLabday AS DATETIME[YY:HH] DEFAULT CURRENTDATE
    CONSTRAINT Labcheck1 CHECK
        (VALUE > DATETIME(1987))
    CONSTRAINT Labcheck2 CHECK
        (WEEKDAY OF VALUE BETWEEN 1 AND 5)
```

Note: The definition of integrity constraint is such that NULL values pass the check in most simple cases. For all examples above, a NULL value yields the result "unknown" for the search condition, thus the negation NOT(..) also yields unknown (and thus the constraint is not violated).

To achieve that a NULL value violates an integrity constraint, the constraint must be formulated like

```
CHECK ( VALUE IS NOT NULL AND ... )
```

3.3 AlterDomainStatement

Serves to alter a domain in the database, i.e. to set or drop a default, to add or remove Check Constraints.

Syntax:

```
AlterDomainStatement ::=
    ALTER DOMAIN DomainName AlterDomainSpec

AlterDomainSpec ::=
```

```
    SET DEFAULT Expression
  | DROP DEFAULT
  | ADD DomainConstraint
  | DROP CONSTRAINT ConstraintName
```

Explanation: Note that no field values in the database are changed by any of these statements.

`SET DEFAULT` sets the default of the domain to the specified value.

`DROP DEFAULT` drops the default value of the domain.

`ADD DomainConstraint` adds the specified domain constraint to the domain. All table fields based on the domain are checked whether they fulfill the new constraint and the statement is rejected if there are any violations against the new constraint.

`DROP CONSTRAINT ConstraintName` drops the specified domain constraint from the domain.

See the *CreateDomainStatement* for the allowed forms of Expression and *DomainConstraint*.

Privileges: The user must be owner of the domain.

Example:

```
ALTER DOMAIN Suppno SET DEFAULT -1
ALTER DOMAIN Suppno DROP DEFAULT
ALTER DOMAIN MyLabday DROP CONSTRAINT Labcheck1
ALTER DOMAIN MyLabday ADD CONSTRAINT Labcheck3
    CHECK (VALUE > DATETIME(1989-2))
```

3.4 DropDomainStatement

Serves to remove a domain from the database.

Syntax:

```
DropDomainStatement ::=
    DROP DOMAIN DomainName DropBehaviour
DropBehaviour ::=
    RESTRICT | CASCADE
```

Explanation: The statement removes the specified domain from the database.

If RESTRICT is specified, the statement is rejected if any field of an existing table is based on the domain or if the domain is used in a CAST expression of any view definition.

If CASCADE is specified, the domain is removed also in the cases where the RESTRICT variant would fail. For all table fields based on the domain, the domain constraints (if any) are integrated as table constraints into the table definitions. The domain default (if any) is integrated as field default unless the field has been specified with an explicit DEFAULT value at table definition time.

Note: The semantics of a DROP ... CASCADE is such that the integrity constraints defined via the domain (if any) effectively are not lost.

Privileges: The user must be owner of the domain.

Example:

```
DROP DOMAIN MyLabday CASCADE
```

3.5 CreateTableStatement

Serves to create a table in the database.

Syntax:

```
CreateTableStatement ::=
    StandardTableStatement|FlatTableStatement|FileTableStatement

StandardTableStatement ::=
    CREATE TABLE TableName [ IkSpec ] [ ClusterSpec ]
    ( TableElem [ , TableElem ] ... )
    [ KeySpec ]

FlatTableStatement ::=
    CREATE FLAT[SizeSpec] TABLE TableName [IkSpec] [ClusterSpec]
    ( TableElem [ , TableElem ] ... )

FileTableStatement ::=
    CREATE FILE(FileName [CodePageSpec] [NullDelimSpec])
```

```
    TABLE WITHOUT IKACCESS
    ( TableElem [ , TableElem ] ... )

SizeSpec ::=
    ( IntegerLiteral [ KB | MB ] )

IkSpec ::=
    { WITH | WITHOUT } IKACCESS

ClusterSpec ::=
    CLUSTER Clusterno [ , [ DELTA ] Startpg ]

ClusterNo ::= IntegerLiteral

Startpg ::= IntegerLiteral

TableElem ::=
    FieldDefinition
  | TableConstraintDefinition

FieldDefinition ::=
    FieldName  DataTypeSpec   [ DefaultClause ]
    [ FieldConstraintDefinition  ] ...

DataTypeSpec ::=
    DataType  |  DomainName

DefaultClause ::=
    DEFAULT  Expression

KeySpec ::= StdKeySpec | HyperCubeKeySpec

StdKeySpec ::=
    KEY IS FieldName [, FieldName ] ... |

HyperCubeKeySpec ::=
    HCKEY [ NOT UNIQUE ] IS FieldName [, FieldName ] ... |
```

Explanation: The *CreateTableStatement* creates a table with the given *TableName*.

The *StandardTableStatement* creates a table as a B-tree. Therefore its data is stored clusterd (sorted) along its primary key specification. This allows efficient lookup of data via the primary key. On the other hand, insertions into sorted data

are complex and therefore costly.

The *FlatTableStatement* creates a table without primary key and without clustering. In contrast to standard tables, data is stored in input order. This allow faster data insertion as data is always appended. Via its *SizeSpec* the table can be restricted to occupy no more than a certain maximum of space. If this maximum is exceeded, the oldest data will automatically be replaced. Thus *Flat Tables* are ideally suited for data staging during bulk load processes, as temporary storage and for logging facilities.

The *FileTableStatement* allows *SpoolFiles* or other compatible file formats to be integrated into the database schema as virtual tables. These `FILE` tables offer read-only access to those files via SQL commands. They can be used throughout SQL `SELECT` statements like any other base relation. The table definition supplies a mapping of columns in the external file to column names and Transbase datatypes. Currently a File table can only be created `WITHOUT IKACCESS` and no key specifications are allowed. Therefore the creation of secondary indexes is currently not possible. These restrictions might be dropped in future Transbase versions. For details on the optional parameters *CodePageSpec* and *NullDelimSpec* please consult the *Table Spool Statement*. FILE tables are primary designed as an advanced instrument for bulk loading data into Transbase and applying arbitrary SQL transformations at the same time.

The *IKSpec* adjusts whether to create a table with or without internal key (IK) access path. IKs are used as row identifier, e.g. for referencing tuples in the base relation after accessing secondary indexes. This IK access path requires additional space of 6 to 8 bytes per tuple. Alternatively Transbase can use the primary key access path. In this case the base table's primary key is stored in all index tuples for referencing the base relation. Depending on how extensive the primary key is, Transbase will automatically decide at table creation time whether to create a table `WITH` or `WITHOUT IKACCESS`. This guarantees optimal space efficiency. If the primary key occupies no more that the IK, then the table is created `WITHOUT IKACCESS`. Else an IK access path is added by default.

Secondary indexes can also be created on *Flat Tables*. As these tables do not have a primary key, secondary indexes are only possible on *Flat Tables* `WITH IKACCESS`. Typically such secondary indexes are added once the load process is complete, so load performance is not compromised by secondary index maintenance.

It is always possible to override this default mechanism of *IKSpec* by adding `WITH` or `WITHOUT IKACCESS` to the create table statement.

The *ClusterSpec* is only important for the development of CD-ROM Databases and explained in the Transbase CD-ROM Database Guide.

Each *FieldDefinition* specifies a field of the table. The ordering of fields is relevant for the `*`-notation in `SELECT * FROM ...`.

TableConstraintDefinition and *FieldConstraintDefinition* are explained in the subsequent chapters.

The *CreateTableStatement* creates a table with the given *TableName*.

In the DEFAULT Expression, no field references nor subqueries are allowed.

Each field has a (explicitly specified or implicit) default value which is taken as input value if a field value is not explicitly specified in an INSERT statement (or by an INSERT via a view). If a DEFAULT clause is specified with an expression evaluating to d, then d is the (explicitly specified) default value for that field, otherwise, if the field is based on a domain with explicit default value d, then d is the default value, otherwise NULL is the default value.

If the key specification is omitted, the combination of all fields implicitly is the key. No column of type BLOB is allowed to be part of the key.

Unless a NOT UNIQUE specification is given, insert and update operations which produce tuples with the same values on all key fields are rejected.

The "KEY IS .." specification creates a table with a compound B-tree index.

The "HCKEY IS .." specification creates a table with a HyperCube index. Key fields of a HyperCube table are restricted to exact arithmetic types (BIGINT, INTEGER, SMALLINT, TINYINT, NUMERIC). If NOT UNIQUE is specified, then also duplicates on the key combination are allowed. NOT UNIQUE, however, is restricted to HyperCube tables. On each HyperCube key field a NOT NULL constraint and a *CheckConstraint* must exist.

The current user becomes owner of the table and gets SELECT-privilege, INSERT-privilege, DELETE-privilege on the table and UPDATE-privilege on all fields of the table. All privileges are grantable.

Note: If there exists one field (or one or more field combinations) which is known to be unique in the table, it is strongly recommended to explicitly specify it as key of the table. One advantage is that uniqueness is guaranteed; another advantage is much better performance in update operations (which normally do not change key values).

Note: The space requirement for a table depends on the order in which the fields are declared. It is advantageous to arrange the fields such that the fixed sized fields precede the variable sized fields. A field is fixed sized if it is not of type CHAR(*), BINCHAR(*), NUMERIC(p,s) and if it is additionally declared NOT NULL. The reason is that all length field pointer in front of the first variable sized field can be suppressed in the storage structure of a tuple.

Privileges: The user must have userclass DBA or RESOURCE. For the definition of userclasses see the chapter *GrantUserclassStatement*, below.

Example:

```
CREATE TABLE quotations
(   suppno        INTEGER       DEFAULT -1 NOT NULL,
    partno        INTEGER       DEFAULT -1 NOT NULL,
    price         NUMERIC (6,2) DEFAULT 0  NOT NULL,
    delivery_time INTEGER,
    qonorder      NUMERIC (4)
)
KEY IS suppno, partno
-- note the two fixed sized fields in front
```

Example:

```
CREATE TABLE persons WITHOUT IKACCESS
(   name  CHAR(*),
    birthday DATETIME [YY:DD],
    talked  MyLabday
)
-- MyLabday is a domain as defined in CreateDomainStatement
-- default value of  "talked" is CURRENTDATE via domain MyLabday
```

Example:

```
CREATE TABLE geopoints
(   info          INTEGER       NOT NULL,
    longitude     NUMERIC(10,7) NOT NULL
      CHECK(longitude BETWEEN -180 AND 180),
    altitude      NUMERIC(9,7)  NOT NULL
      CHECK(altitude BETWEEN -90 AND 90)
)
HCKEY IS longitude, altitude
```

3.5.1 TableConstraintDefinition, FieldConstraintDefinition

Overview syntax for specification of integrity constraints in a *CreateTableStatement*.

Syntax:

```
TableConstraintDefinition ::=
   [ CONSTRAINT  ConstraintName ]  TableConstraint

FieldConstraintDefinition ::=
   [ CONSTRAINT  ConstraintName ]  FieldConstraint

TableConstraint ::=
   PrimaryKey
 | CheckConstraint
 | ForeignKey

CheckConstraint ::=
   CHECK (SearchCondition)

ForeignKey ::=
   FOREIGN KEY (FieldNameList)  ReferencesDef

ReferencesDef ::=
   REFERENCES  TableName  [ (FieldNameList) ]
   [ ON DELETE   Action ]
   [ ON UPDATE  NO ACTION ]

Action ::=
   NO ACTION  |  CASCADE  |  SET DEFAULT  |  SET NULL

FieldConstraint ::=
   NOT  NULL
 | PRIMARY  KEY
 | CheckConstraint
 | ReferencesDef
```

Explanation: Explanations are given in the subsequent separate chapters.

The construct FieldConstraint is subsumed by the more general TableConstraint. In certain special cases, the syntactic variant FieldConstraint allows a more compact notation for a TableConstraint. There are no performance differences with the 2 notations.

Note: All constraints are effectively checked after execution of each SQL query.

3.5.2 PrimaryKey

Specify the main key for a table.

Syntax:

```
PrimaryKey ::= StdKeySpec | HyperCubeKeySpec

StdKeySpec ::=
    PRIMARY  KEY  (FieldNameList)

HyperCubeKeySpec ::=
    PRIMARY  HCKEY [NOT UNIQUE] (FieldNameList)
```

Explanation: Only one PrimaryKey specification is allowed per table definition.

If no PrimaryKey is specified, all fields in their natural order form the primary key.

The SQL-2 formulation `PRIMARY KEY (f1, f2, ..,fn)` is equivalent to the alternative (Transbase proprietary) formulation `KEY IS f1,f2,..,fn` (see below for an example). The SQL-2 formulation `PRIMARY HCKEY [NOT UNIQUE](f1, f2, ..,fn)` is equivalent to the alternative (Transbase proprietary) formulation `HCKEY [NOT UNIQUE] IS f1,f2,..,fn`

For the semantics of the key specification see *CreateTableStatement*. See also the Performance Guide for more details.

Example: The following two examples are equivalent. The first is the official SQL-2 notation supported by Transbase, the second is an alternative notation also supported by Transbase (note that the formulations exclude each other):

```
CREATE  TABLE  quotations
(   suppno        INTEGER,
    partno        INTEGER,
    price         NUMERIC(9,2),
    delivery_time INTEGER,
    PRIMARY KEY   (suppno, partno)
)

CREATE  TABLE  quotations
(   suppno        INTEGER,
    partno        INTEGER,
    price         NUMERIC(9,2),
    delivery_time INTEGER
 )  KEY  IS suppno, partno
```

Example: The following two examples show alternative formulations of primary key via a TableConstraint and a FieldConstraint - this is possible if and only if one single field constitutes the primary key:

```
CREATE  TABLE  suppliers
(    suppno  INTEGER,
     name    CHAR(*),
     address CHAR(*),
     PRIMARY KEY(suppno)
)

CREATE  TABLE  suppliers
(    suppno  INTEGER  PRIMARY KEY,
     name    CHAR(*),
     address CHAR(*)
)
```

3.5.3 CheckConstraint

Specify a CheckConstraint for a table.

Syntax:

```
CheckConstraint ::=
    CHECK (SearchCondition)
```

Explanation: The SearchCondition specifies an integrity condition which must be fulfilled for all tuples of the table.

In detail, for all tuples of the table which are inserted or updated an error is reported if the condition `NOT (SearchCondition)` evaluates to TRUE.

If the CheckConstraint is specified with an explicit *ConstraintName*, an integrity violation message concerning this CheckConstraint reports this name, otherwise an implicitly generated name is reported. For the sake of easy error analysis, it is thus recommended to specify explicit and self-explanatory constraint names.

Example:

```
CREATE  TABLE  quotations
(    suppno        INTEGER,
     partno        INTEGER,
```

```
     price         NUMERIC(9,2),
     delivery_time INTEGER,
     CONSTRAINT    price100 CHECK (price < 100)
)

CREATE  TABLE  quotations
(    suppno        INTEGER,
     partno        INTEGER,
     price         NUMERIC(9,2),
     delivery_time INTEGER,
     CONSTRAINT    price_deliv
     CHECK (price < 20 OR delivery_time < 3)
)
```

In the first example, only one field is involved. Therefore it can also be formulated using the syntactic variation FieldConstraint:

```
CREATE  TABLE  quotations
(    suppno        INTEGER,
     partno        INTEGER,
     price         NUMERIC(9,2)
     CONSTRAINT    price100 CHECK (price < 100),
     delivery_time INTEGER
)
```

Note that in a FieldConstraint, there is no comma between the field definition and the constraint definition.

Catalog Tables: One entry into the table "sysconstraint" is made: for each constraint which is not a referential constraint.

Note: The definition of integrity violation is such that NULL values pass the test in most cases. In the example above, the constraint "price100" is not violated by a tuple with a NULL value on price, because the SearchCondition evaluates to unknown (thus the negation NOT(..) also evaluates to unknown but not to TRUE).

To make NULL values fail the test, one must explicitly formulate the CheckConstraint like: `CHECK (price IS NOT NULL AND ...)`.

Here, one also can use the shorthand notation NOT NULL:

```
CREATE  TABLE  quotations
```

```
(   suppno        INTEGER,
    partno        INTEGER,
    price         NUMERIC(9,2)  NOT NULL
    CONSTRAINT    price100 CHECK (price < 100),
    delivery_time INTEGER
)
```

3.5.4 ForeignKey

Specify a Referential Constraint between 2 tables.

Syntax:

```
ForeignKey ::=
    FOREIGN KEY (FieldNameList)  ReferencesDef

ReferencesDef ::=
    REFERENCES  TableName  [ (FieldNameList) ]
    [ ON DELETE   Action ]
    [ ON UPDATE   Action ]

Action ::=
    NO ACTION  |  CASCADE  |  SET DEFAULT  |  SET NULL
```

Explanation: A referential constraint between 2 tables is specified.

With respect to the constraint, the table containing the fields of the foreign key is called the referencing table, the table which is mentioned after REFERENCES is called the referenced table. Analogously, the fields in the FOREIGN KEY clause and the (explicit or implicit) fields in the REFERENCES clause are called referencing fields and referenced fields, resp. The referencing and referenced fields must have same number and identical types.

If no field name list is specified in the REFERENCES clause, then the primary key combination of the referenced table constitutes the referenced fields. The referenced fields either must constitute the primary key or must have a UNIQUE INDEX.

The referential constraint is as follows:

For each tuple in the referencing table whose referencing fields do not have any NULL value, there must be one tuple in the referenced table with identical field values on the corresponding referenced fields.

Let RG and RD the referencing table and the referenced table, resp., i.e. RG references RD.

The following statements potentially violate a referential constraint:

1. INSERT, SPOOL, UPDATE, in RG
2. DELETE, UPDATE in RD.

A referential constraint can be specified to trigger compensating actions.

Specification of NO ACTION effectively is the absence of a triggered action.

If CASCADE is specified:

A deletion of tuple t in RD triggers the deletion of all matching tuples in the RG (thus maintaining the referential constraint). An update of a referenced field in RD triggers the corresponding update of all referencing fields in RG to the same value (thus maintaining the referential constraint).

If SET NULL or SET DEFAULT is specified:

A deletion of tuple t in RD triggers the update of the referencing fields of all matching tuples in RG to NULL or their DEFAULT value. The first case always maintains the referential constraint, the second case only if there is a matching DEFAULT value tuple in RD. An update is handled analogously.

Example:

```
CREATE  TABLE  quotations
(   suppno        INTEGER,
    partno        INTEGER,
    price         NUMERIC(9,2),
    delivery_time INTEGER,
    CONSTRAINT    quotrefsupp
        FOREIGN KEY (suppno) REFERENCES suppliers(suppno)
        ON DELETE  SET NULL,
    CONSTRAINT quotrefpart
        FOREIGN KEY (partno) REFERENCES parts(partno)
        ON DELETE  CASCADE,
)
```

In this (single field reference) example, also the syntactic shorthand variant of FieldConstraint can be used as shown below:

```
CREATE  TABLE  quotations
(   suppno        INTEGER
        CONSTRAINT quotrefsupp
        REFERENCES suppliers(suppno)
```

```
        ON DELETE  SET NULL,
    partno        INTEGER
        CONSTRAINT quotrefpart
        REFERENCES parts(partno)
        ON DELETE  CASCADE,
    price         NUMERIC(9,2),
    delivery_time INTEGER
)
```

Catalog Tables: Entries into the table "sysrefconstraint" are made: for a ref. constraint with a n-ary field combination, n tuples are inserted into sysrefconstraint.

Important Note on Performance: DELETE and UPDATE operations on referenced tables which require the referential check on the referencing table are slow if the referencing table has not a secondary index (or the primary key) on the referencing fields.

On the contrary, INSERTs and UPDATEs on referencing fields requiring the referential check on the referenced table are fast because by definition there is a index on the referenced fields.

Note: Like all constraints, referential constraints are effectively checked after execution of each SQL query. In general, it is therefore not possible to insert tuples into tables in arbitrary order if there exists a referential constraint between them.

3.6 AlterTableStatement

Serves to alter fields of a table and to add or remove table constraints.

Syntax:

```
AlterTableStatement ::=
    AlterTableAddField
    AlterTableModifyField
    AlterTableChangeField
    AlterTableConstraint
    AlterTableMove
```

Privileges: The user must be owner of the table.

3.6.1 AlterTableAddField

Serves to add a field to a table.

Syntax:

```
AlterTableAddField ::=
    ALTER TABLE  TableName
        ADD FieldDefinition

FieldDefinition ::= see CreateTableStatement
```

Explanation: A new field is added as the last field of the table and initialized with the explicitly specified or implicit default value.

Example:

```
ALTER  TABLE  quotations ADD comment CHAR(*) DEFAULT ''
```

3.6.2 AlterTableModifyField

Serves to modify a field of a table.

Syntax:

```
AlterTableModifyField ::=
    ALTER TABLE  TableName
        MODIFY FieldDefinition

FieldDefinition ::= see CreateTableStatement
```

Explanation: The data type of an existing field is changed to the specified type. Already existing data will be converted due to this modification. Note that not all data types are compatible among each other.

Example:

```
ALTER  TABLE  quotations MODIFY delivery_time DOUBLE
```

3.6.3 AlterTableChangeField

Serves to alter a field of a table.

Syntax:

```
AlterTableChangeField ::=
    ALTER TABLE  TableName
        ALTER [COLUMN] FieldName
        DefaultAction

DefaultAction ::=
        SET  DEFAULT  Expression
      | DROP DEFAULT
```

Explanation: The SET DEFAULT variant specifies a default value for the field.

The DROP DEFAULT removes the default value from the field.

Both statements do not change field values in the database.

Note: There is a slight difference between SET DEFAULT NULL and DROP DEFAULT : The former leaves the field with an explicit default which overrides the default of an underlying domain (if any), the latter removes an explicit default, thus the default (if any) of the underlying domain (if any) becomes effective.

Example:

```
ALTER  TABLE  quotations ALTER price SET DEFAULT  100.0
ALTER  TABLE  quotations ALTER delivery_time  DROP DEFAULT
```

3.6.4 AlterTableConstraint

Serves to add or remove a table constraint.

Syntax:

```
AlterTableConstraint ::=
    ALTER TABLE  TableName  ConstraintAction

ConstraintAction ::=
    ADD  TableConstraintDefinition
  | DROP  CONSTRAINT  ConstraintName
```

Explanation: TableConstraintDefinition is defined in the corresponding chapter. All except the redefinition of PRIMARY KEY is allowed on this position.

The ADDition of a table constraint is rejected if the values in the database do not fulfill the constraint.

Example:

```
ALTER  TABLE  quotations DROP CONSTRAINT quotrefpart

ALTER  TABLE  parts ADD  CONSTRAINT  qonh
    CHECK (quonhand/10*10 = quonhand)

ALTER  TABLE  quotations ADD  CONSTRAINT  quotrefpart
    FOREIGN KEY (partno) REFERENCES  parts2
    ON DELETE CASCADE
```

3.6.5 AlterTableMove

Serves to reorganize a table.

Syntax:

```
AlterTableMove ::=
    ALTER TABLE  TableName
        MOVE [ MoveElem [, MoveElem ] ... ] [ TO ] BLOCK BlockNo

MoveElem ::=
    TABLE | BLOBCONTAINER | INDEX IndexName

BlockNo ::= IntegerLiteral
```

Explanation: The logical page ordering of segments (tables, indices or blobcontainers) can be reorganized by this statement. The lower bound address of the reorganized segment is specified by BlockNo, the upper bound address is given by the maximal database size.

If no move element is specified, then all indices of the table, the blobcontainer (if the table contains one or more blob fields) and the table itself are moved.

This statement is only allowed on Transbase Standard Databases.

Example:

```
ALTER TABLE quotations MOVE TABLE, INDEX quot_pa_pr TO BLOCK 1000;

ALTER TABLE quotations MOVE BLOCK 50000;
```

3.7 DropTableStatement

Serves to drop a table in the database.

Syntax:

```
DropTableStatement ::=
    DROP TABLE TableName
```

Explanation: The specified table, all indexes and all triggers defined on that table are dropped. All views which are directly or transitively based on the specified table are also dropped.

Privileges: The current user must have userclass DBA or must be owner of the table.

Example:

```
DROP TABLE quotations
```

3.8 CreateViewStatement

Serves to create a view in the database.

Syntax:

```
CreateViewStatement ::=
    CREATE VIEW ViewName
    [ ( FieldNameList ) ]
    AS SelectStatement
    [ WITH CHECK OPTION ]

FieldNameList  ::=
    FieldName [, FieldName ] ...
```

Explanation: The CreateViewStatement creates a view with the specified ViewName and FieldName(s).

If no fieldlist is specified then the derived names of the SelectStatment implicitly form the field list. If an element of the SELECT list has no derived name (expression) then an error is returned. Note that by use of the AS clause, each SELECT element can explicitly be given a name.

At any time, the contents of the view is the actual result of the defining SelectStatement.

There must not be a table or view with the same name in the database. An n-ary view must be defined by a SelectStatement which delivers n-ary tuples.

SelectStatements are explained in the 'Data Modification Language (DML)' portion of this manual.

The created view is updatable if the SelectStatement is updatable. If the WITH CHECK OPTION is specified, the view must be updatable.

If the WITH CHECK OPTION is specified for a view v, then Insert and Update operations are rejected whenever any inserted or updated tuple does not fulfill the SearchCondition of the defining SelectStatement of v or any other view on which v is transitively based.

A view can be used in any SelectStatement like a table. Especially, existing views can be used for the definition of a view.

Insert, Update, Delete are only allowed on updatable views.

Indexes on views are not allowed.

The current user becomes owner of the view. If the view is not updatable, the user only gets a non-grantable SELECT-privilege on the view, otherwise the user gets the same view privileges on the view as those on the (one and only) table or view which occurs in the defining SelectStatement.

A view may also contain one ore more *RemoteTableNames*. When evaluating remote views, the privileges of the view owner apply for accessing remote tables. However, the current user must have at least ACCESS privilege for the remote database. If a updatable remote view is is specified as the target of an UPDATE or DELETE operation, all subqueries (if any) must specify tables residing on the same database. However, if the target table is local, any tables (remote or local) may be specified in subqueries.

Privileges: The current user must have userclass DBA or RESOURCE and must have the privileges for the defining *SelectStatement* (see *SelectStatement*).

Note: For a view, the constituting tuples are not stored in the database, but only the information of the view definition. Therefore, also simple queries on views normally require the execution time of the defining statement.

Example:

```
CREATE VIEW totalprice (supplier, part, total)
AS
SELECT name, description, price * qonorder
FROM suppliers, quotations, inventory
WHERE suppliers.suppno = quotations.suppno
    AND inventory.partno = quotations.partno
    AND qonorder > 0
-- a non-updatable view
```

3.9 DropViewStatement

Serves to drop a view in the database.

Syntax:

```
DropViewStatement :=
    DROP VIEW ViewName
```

Explanation: The specified view is dropped. All views which are directly or transitively based on the specified view are also dropped.

Privileges: The current user must have userclass DBA or must be owner of the view.

Example:

```
DROP VIEW totalprice
```

3.10 CreateIndexStatement

Serves to create an index, fulltext index or a bitmap index on a table.

Syntax:

```
CreateIndexStatement ::= StandardIndexStatement |
    FulltextIndexStatement | BitmapIndexStatement
```

3.10.1 StandardIndexStatement

Serves to create a (non-fulltext) index on a table.

Syntax:

```
StandardIndexStatement ::=
    CREATE [UNIQUE] INDEX IndexName
    ON TableName (FieldNameList)
    [ HyperCubeKeySpec ]

FieldNameList :: =
    FieldName [, FieldName] ...

HyperCubeKeySpec ::=
    HCKEY [NOT UNIQUE] IS KeyList

KeyList ::= FieldNameList
```

Explanation: An index with the specified name is created on the specified fields of the specified table. There must not be an index with the same name on any table in the database. There must not be an index on the same fields (in the order specified) of the table. No FieldName may occur twice in the FieldNameList.

If no *HyperCubeKeySpec* is given then the index is created as a standard compound key B-tree index. The fields specified in the FieldNameList constitute the key of the B-tree. Efficient search with all keys or a prefix of the keys in that order then is supported. If "CREATE UNIQUE .." is specified then the field combination is required to be unique and insert and update operations which would result in at least two tuples with the same values on the specified fields are rejected.

With the *HyperCubeKeySpec* clause and the HCKEY specification, a HyperCube tree can be specified instead of a standard compound B-tree. A HyperCube tree should have no fields as part of key which typically are not searched for - therefore fields and keys can be specified separately in this case. The specified keys are UNIQUE by default unless the "NOT UNIQUE" clause is specified. The "CREATE UNIQUE .." formulation is not permitted if a *HyperCubeKeySpec* is given. All fields used as HyperCube key fields must be NOT NULL and must have a range check constraint in the underlying base table.

Indexes have no effect on query results except for possibly different performance (depending on the query type) and possibly different sort orders of query results (if no ORDER BY-clause is specified in the query).

Indexes on views are not allowed.

Indexes on tables created WITHOUT IKACCESS are not allowed.

BLOB fields can only be indexed by fulltext indexes.

Note: It is unwise to create a standard B-tree index on the highest weighted key fields because in Transbase an unnamed (multi attribute) index exists on the key fields anyway.

Privileges: The current user must have userclass DBA or must have userclass RESOURCE and be owner of the table.

Example:

```
CREATE INDEX quot_pa_pr
ON quotations (partno,price)
```

3.10.2 FulltextIndexStatement

Serves to create a fulltext index on a VARCHAR or CHAR or BLOB field of a table.

Syntax:

```
FulltextIndexStatement ::=
    CREATE [POSITIONAL] FULLTEXT INDEX IndexName
    [FulltextSpec] ON TableName (FieldName)
    [ScratchArea]

FulltextSpec ::=
    [ { Wordlist | Stopword} ]  [ Charmap]  [ Delimiters ]

Wordlist ::=
    WORDLIST  FROM  WTableName

Stopword ::=
    STOPWORDS  FROM  STableName

Charmap ::=
    CHARMAP  FROM  CTableName
```

```
Delimiters ::=
    DELIMITERS FROM  DTableName
  | DELIMITERS  NONALPHANUM

WTableName ,STableName ,CTableName ,DTableName ::=
    Identifier

ScratchArea ::=
    SCRATCH IntegerLiteral MB
```

Explanation: All explanations are given in the separate chapter on Fulltext Indexes.

3.10.3 BitmapIndexStatement

Serves to create a bitmap index on a `BOOL`, `TINYINT`, `SMALLINT` or `INTEGER` field of a table.

Syntax:

```
BitmapIndexStatement ::=
    CREATE BITMAP INDEX IndexName
    ON TableName (FieldName)
```

Explanation: A bitmap index with the specified name is created on the specified field of the specified table. There must not be an index with the same name on any table in the database. There must not be an index on the same field of the table.

Bitmap indexes are preferably used for non-selective columns having few different values (e.g. classifications). Bitmap indexes are innately very space efficient. With their additional compression in average they occupy less than on bit per index row. A bitmap index can be created on any base relation (B-Tree or Flat) having a single `INTEGER` field as primary key or an `IKACCESS` path.

Bitmap processing allows inexpensive calculation of logical combinations (`AND`/ `OR`/ `NOT`) of restrictions on multiple non-selective fields using bitmap intersection and unification.

3.11 DropIndexStatement

Serves to drop an index.

Syntax:

```
DropIndexStatement ::=
    DROP INDEX IndexName
```

Explanation: The specified index is dropped.

Privileges: The current user must have userclass DBA or must be owner of the table on which the index has been created.

Example:

```
DROP INDEX quot_pa_pr
```

3.12 CreateTriggerStatement

Serves to create a trigger on a table.

Syntax:

```
CreateTriggerStatement ::=
    CREATE TRIGGER TriggerName
    TriggerActionTime TriggerEvent
    ON TableName
    [ REFERENCING OldNewAliasList ]
    TriggeredAction

TriggerActionTime ::=
    BEFORE | AFTER

TriggerEvent ::=
    INSERT | DELETE | UPDATE [ OF FieldNameList ]

FieldNameList :: =
    FieldName [, FieldName] ...

OldNewAliasList ::=
    OldNewAlias [ OldNewAlias ]

OldNewAlias ::=
```

```
    OLD [ ROW ] [ AS ] CorrelationName
 |  NEW [ ROW ] [ AS ] CorrelationName

TriggeredAction ::=
    [ FOR EACH { ROW | STATEMENT } ]
    [ WHEN ( SearchCondition ) ]
    TriggerSqlStatement

TriggerSQLStatement ::=
    DMLorCallStatement
 |  BEGIN ATOMIC DMLorCallStatement [; DMLorCallStatement ] ... END

DMLorCallStatement ::=
   InsertStatement
 | UpdateStatement
 | DeleteStatement
 | CallStatement
```

Explanation: A trigger is a user defined sequence of SQL modification statements or CallStatements which is automatically executed when a INSERT, UPDATE or DELETE statement is executed. The specification of a trigger contains a triggername, a triggerevent on a table (e.g. INSERT ON quotations) which specifies when the trigger is to be fired, a detailed trigger action time (BEFORE or AFTER) which specifies whether the trigger has to fired before or after the insert action. Furthermore, a trigger has to specified to be either a row trigger (FOR EACH ROW, i.e. to be fired once for each processed tuple) or a statement trigger (FOR EACH STATEMENT, i.e. to be fired only once for the complete modification statement).

For a row trigger, the specified actions may refer to the actually processed tuple values. The syntax NEW.fieldname is the value of fieldname of the inserted tuple or the (possibly changed) value of fieldname for an tuple being updated. The syntax OLD.fieldname is the value of fieldname of a deleted tuple or the original field value for an tuple being updated.

The firing of a row trigger may be restricted along a condition (SearchCondition) which also may refer to the NEW or OLD field values of the tuple currently processed.

The keywords NEW and OLD may be overridden by a OldNewAliasList.

When a trigger is fired it runs under the privileges of the creator of the trigger.

If more than one trigger qualifies at the same TriggerActionTime, the order of execution is defined by ascending creation date.

UPDATEs and DELETEs on a table T which has triggers and also is the target of a referential constraint require special consideration. Referential actions (called reference triggers here) may occur if the reference constraint is specified with CASCADE, SET NULL or SET DEFAULT.

Triggers are performed in the following order:

```
(1) Before-StatementTriggers
(2) Before-Row Triggers
(3) Reference Triggers
(4) After-Row Triggers
(5) After-Statement Triggers
```

Note that the firing of a trigger may cause the firing of subsequent triggers. It is recommended to use triggers moderately to keep the complexity of nested actions small. In particular, it is strongly discouraged to construct a set of triggers which lead to the effect that a modification of a table T fires a trigger which transitively fires another trigger which also tries to modify table T. This may cause endless loops or nonpredictable effects.

Privileges: The current user must have userclass DBA or RESOURCE and becomes the owner of the trigger. Additionally, the current user must have SELECT privilege on the table on which the trigger is created.

Example:

```
CREATE TRIGGER quotations_upd
BEFORE UPDATE OF price ON quotations
FOR EACH ROW
WHEN (NEW.price > OLD.price )
INSERT INTO logquotationsprice
VALUES (NEW.suppno,NEW.partno,NEW.price-OLD.price)
```

Example:

```
CREATE TRIGGER quotations_ins
BEFORE INSERT ON quotations
FOR EACH ROW
CALL JavaFuncQuot(NEW.suppno,NEW.partno,NEW.price)
```

3.13 DropTriggerStatement

Serves to drop a trigger.

Syntax:

```
DropTriggerStatement ::=
    DROP TRIGGER TriggerName
```

Explanation: The specified trigger is dropped. Note that the trigger also is dropped if the table is dropped on which the trigger is defined.

Privileges: The current user must have userclass DBA or must be owner of the trigger.

Example:

```
DROP TRIGGER quotations_upd
```

3.14 CreateSequenceStatement

Creates a sequence.

Syntax:

```
CreateSequenceStatement ::=
    CREATE SEQUENCE SequenceName SeqOptions
SeqOptions ::= [ StartSpec ] [ IncrSpec ] [ MaxSpec ] [ CYCLE ]
StartSpec ::= START [WITH] IntegerLiteral
IncrSpec ::= INCREMENT [BY] IntegerLiteral
MaxSpec ::= MAXVALUE IntegerLiteral
```

Explanation: Creates a sequence with the specified name. A sequence is an object which can be used to generate increasing numbers of type Integer. These numbers are unique even if generated by concurrent transactions. Also no lock conflicts arise due to the use of sequences. For a sequence S, there are 2 operations available namely S.nextval and S.currval. The first S.nextval operation delivers the value specified in StartSpec or 1 as default. Each S.nextval increases the value of S by the value specified in IncrSpec or 1 as default. If a MaxSpec has been given and the nextval operation would generate a value beyond the maximal value then either an error is generated or (if CYCLE has been specified) the startvalue again is delivered as next value. S.nextval also is permitted as default specification for a field of a table. The S.currval operation is only allowed if there has been a S.nextval operation within the same transaction and again delivers the last value delivered by S.nextval. S.currval does not increase the current value of S.

Privileges: To create a sequence, the current user must have userclass DBA or RESOURCE. For nextval the user must have UPDATE privilege on the sequence. For currval the user must have SELECT privilege on the sequence. Privileges on sequences are granted and revoked like those on tables.

Example:

```
CREATE SEQUENCE S START 1 INCREMENT 2
```

3.15 DropSequenceStatement

Drops a sequence.

Syntax:

```
DropSequenceStatement ::=
    DROP SEQUENCE SequenceName
```

Explanation: Drops the sequence with the specified name.

Privileges: The current user must be owner of the sequence.

Example:

```
DROP SEQUENCE S
```

3.16 GrantUserclassStatement

Serves to install a new user or to raise the userclass of an installed user.

Syntax:

```
GrantUserclassStatements ::=
    GRANT Userclass TO UserName

Userclass ::=
    ACCESS
  | RESOURCE
  | DBA
```

Explanation: If the specified user is not yet installed then the statement installs the user and sets its userclass to the specified one.

If the specified user is already installed then the statement raises its userclass to the specified one. In the latter case the specified userclass must be of higher level than the user's current userclass.

The userclass of a user defines the permission to login to the database and to create objects (tables, views, indexes). The special userclass DBA serves to grant the privileges of a superuser who has all rights.

Userclass	Level	Description
DBA	3	All rights
RESOURCE	2	Can create
ACCESS	1	CANNOT create
NO_ACCESS	0	Cannot login

Privileges: The current user must have userclass DBA.

Example:

```
GRANT RESOURCE TO charly
GRANT ACCESS TO jim
```

3.17 RevokeUserclassStatement

RevokeUserclassStatement

Serves to lower the userclass of an installed user or to disable a user from login.

Syntax:

```
RevokeUserclassStatement ::=
    REVOKE ACCESS FROM Username [ CASCADE ]
  | REVOKE RESOURCE FROM Username
  | REVOKE DBA FROM Username
```

Explanation: The specified user must be an installed user with a userclass which is not lower than the specified one. The statement lowers the userclass of the user to the level immediately below the specified userclass.

In particular, if userclass ACCESS is revoked the user cannot login to the database anymore. If CASCADE is specified, all tables and domains owned by the user are

also deleted (domain information used by other tables is expanded like in DROP DOMAIN .. CASCADE). If CASCADE is not specified, all tables and domains owned by the user remain existent and their ownership is transferred to tbadmin.

3.18 GrantPrivilegeStatement

Serves to transfer privileges to other users.

Syntax:

```
GrantPrivilegeStatement ::=
    GRANT Privileges ON TableName
    TO UserList [WITH GRANT OPTION]

Privileges ::=
    AllPrivileges
  | PrivilegeList

AllPrivileges ::=
    ALL [ PRIVILEGES ]

PrivilegeList ::=
    Privilege [, Privilege] ...

Privilege ::=
    SELECT
  | INSERT
  | DELETE
  | UPDATE [ (FieldNameList) ]

UserList ::=
    UserIdent [, UserIdent] ...

UserIdent ::=
    PUBLIC | UserName
```

Explanation: The specified privileges on the table are granted to the specified user; the privileges of the user issuing the statement remain unchanged.

If the WITH GRANT OPTION is specified, the privileges are grantable, i.e. the users get the right to further grant the privileges, otherwise not.

The table may be a base table or a view.

The variant AllPrivileges is equivalent to a PrivilegeList where all four Privileges are specified and where all FieldNames of the table are specified in the UPDATE-privilege.

A missing FieldNameList is equivalent to one with all fields of the table.

If PUBLIC is specified then the privileges are granted to all users (new users also inherit these privileges).

The maximum number of users in the UserList is 50.

Note: A privilege can be granted both to a specific user and to PUBLIC at the same time. To effectively remove the privilege from the user, both grantings must be revoked.

UpdatePrivileges can be granted on field level whereas SELECT-privileges cannot. To achieve that effect, however, it is sufficient to create an appropriate view and to grant SELECT-privilege on it.

Privileges: The current user must have userclass DBA or must have all specified privileges with the right to grant them.

Example:

```
GRANT SELECT, UPDATE (price, qonorder)
    ON quotations TO jim,john
GRANT SELECT ON suppliers
    TO mary WITH GRANT OPTION
```

3.19 RevokePrivilegeStatement

Serves to revoke privileges which have been granted to other users.

Syntax:

```
RevokePrivilegeStatement ::=
    REVOKE Privileges ON TableName
    FROM UserList

Privileges ::=
    AllPrivileges
  | PrivilegeList
```

```
AllPrivileges ::=
    ALL [ PRIVILEGES ]

PrivilegeList ::=
    Privilege [, Privilege] ...

Privilege ::=
    SELECT
  | INSERT
  | DELETE
  | UPDATE [ (FieldNameList) ]

UserList ::=
    UserIdent [, UserIdent] ...

UserIdent ::=
    PUBLIC | UserName
```

Explanation: If the current user is owner of the table, then the specified privileges are removed from the user such that none of the privileges are left for the user.

If the current user is not owner of the table, then the privilege instances granted by the current user are removed from the specified users. If some identical privileges had been additionally granted by other users, they remain in effect (see Example).

If a SELECT privilege is revoked then all views which depend on the specified table and are not owned by the current user are dropped. If an INSERT or UPDATE or DELETE privilege is revoked then all views which depend on the specified table and are not owned by the current user and are updatable are dropped.

Note: It is not an error to REVOKE privileges from a user which had not been granted to the user. This case is simply treated as an operation with no effect. This enables an error-free REVOKING for the user without keeping track of the granting history.

Example: User jim:

```
CREATE TABLE jimtable ...
GRANT SELECT ON jimtable TO mary, anne WITH GRANT OPTION
```

User mary:

```
GRANT SELECT ON jimtable TO john
```

User anne:

```
GRANT SELECT ON jimtable TO john
```

If owner jim then says:

```
REVOKE SELECT ON jimtable FROM john
```

then john looses SELECT-privilege on jimtable
If, however, anne or mary (but not both) say:

```
REVOKE SELECT ON jimtable FROM john
```

then john still has SELECT-privilege on jimtable.

3.20 AlterPasswordStatement

Serves to install or change a password.

Syntax:

```
AlterPasswordStatement ::=
    ALTER PASSWORD FROM Oldpassword TO Newpassword

Oldpassword ::=
    CharLiteral

Newpassword ::=
    CharLiteral
```

Explanation: Oldpassword must match the password of the current user. The password is changed to Newpassword.

After installing a user the password is initialized to the empty string.

Note: Only the first eight characters of passwords are significant.

Example:

```
ALTER PASSWORD FROM '' TO 'xyz'
ALTER PASSWORD FROM 'xyz' TO 'acb'
```

Chapter 4

Data Import

Mass data import from various data sources is supported in Transbase. Data can origin from external data sources such as databases, from flat files storing delimiter separated values and from XML files.

There are two spool statements with the following functions:

- transfer of external data from a file into the database (*SpoolTable*)
- transfer of a query results into a text file (*SpoolFile*).

The first command is useful for building up a database from external data (residing on textfiles in a standard format, see below). The latter command is for extracting data from the database into textfiles. Also some Transbase tools like tbarc (the Transbase Archiver) use the facilities of the Spooler.

Also BLOB objects (binary large objects) can be handled by the spooler although - of course - the corresponding files then do not contain text in general.

There is the possibility to choose between the DSV and the XML mode of the data spooler. Both modes are explained next.

4.1 The DSV Spooler

The DSV Spooler (delimiter separated values) works with quite simple text documents as spool files. This means, that each tuple needs to have a value for each column of the destination table. Furthermore, the ordering of the tuple fields and the table columns have to be same. More details about the DSV spool files can be found in the section 4.1.3.

4.1.1 SpoolTableStatement, SpoolFileStatement

Syntax:

```
SpoolTableStatement ::=
   SPOOL TableName FROM [SORTED] FileName [LOCAL]
     CodePageSpec NullDelimSpec

SpoolFileStatement ::=
   SPOOL INTO FileName NullDelimSpec SelectStatement

NullDelimSpec ::=
   [ NullSpec ]  [ DelimSpec ]
 | [ DelimSpec ] [ NullSpec ]

 NullSpec ::=
   NULL [ IS ] StringLiteral

CodePageSpec ::=
   [ CODEPAGE [IS] CodePage [ [WITH | WITHOUT] PROLOGUE ] ]

CodePage ::=
   UTF8 | UCS | UCS2 | UCS4 | UCS2LE | UCS2BE | UCS4LE | UCS4BE

DelimSpec ::=
   DELIM [ IS ] { TAB | StringLiteral }
```

Explanation: *FileName* is an identifier for the file.

The syntax of *FileName* is defined via the underlying operating system.

The *SpoolTableStatement* inserts tuples (records) from the specified file into the specified table (base table or view). The specified table must exist (but need not necessarily be empty). Thus, the *SpoolTableStatement* can be used as a fast means for insertion of bulk data.

The *SpoolTableStatement* has very good performance if the records in the table are ascendingly sorted by the table key(s) (best performance is achieved if the table additionally is empty). If the records are not sorted then Transbase inserts on-the-fly those records which fulfill the sortorder, the others are collected, then sorted, then inserted. For very large unsorted spoolfiles, it can be advantageous to split and spool them into pieces depending on the available disk space which is additionally needed temporarily for sorting.

Usage of the keyword "SORTED" allows to test if the input is actually sorted (if specified, an error is reported and the TA is aborted when a violation of the sort

order is detected). This feature does not influence the spool algorithm but only checks if the input was suited to be spooled with maximal performance.

Without the LOCAL keyword, the specified file is read by the client application and transferred to the tbkernel process. If the file is accessible by the tbkernel process, the LOCAL clause can be used to speed up the spool process: in this case the tbkernel process directly accesses the file under the specified name which must be a complete path name.

For the checking of integrity constraints (keys, null values) the same rules as in *InsertStatement* (see chapter 5.16 below) apply.

The *SpoolFileStatement* stores the result tuples of the specified *SelectStatement* into the specified file (which is created if it does not yet exist, overwritten otherwise).

The spool files are searched or created in the current directory by default if they are not absolute pathnames.

For C programmers at the `tbx`interface a special request SET_DAT_DIR is available to change the default.

The *StringLiteral* in a *NullSpec* and *DelimSpec* (if specified) must be of type CHAR(1) or BINCHAR(1), i.e. of byte length 1. See chapter 4.1.3 (External File Format) for the meaning of these specifications.

Transbase V5.3: The codepage specification UTF8 means that the external file is UTF8-coded.

The codepage specification UCS2LE means that the external file is UCS2 (2 byte fixed length, little-endian). The codepage specification UCS2BE means that the external file is UCS2 (2 byte fixed length, big-endian). The codepage specification UCS2 means that the external file is UCS2 (2 byte fixed length, default format).

The codepage specification UCS4LE means that the external file is UCS4 (4 byte fixed length, little-endian). The codepage specification UCS4BE means that the external file is UCS4 (4 byte fixed length, big-endian). The codepage specification UCS4 means that the external file is UCS4 (4 byte fixed length, default format).

The codepage specification UCS means that the external file is the default UCS in default format. which is e.g. UCS2 on Windows platforms and UCS4 on UNIX platforms.

The optional PROLOGUE clause can be applied if the external file is prologued with the Unicode character 0uFEFF. If no PROLOGUE clause is given on input and no byte-order is specified, the byte order is determined automatically. If a byte-order is specified, and a differing PROLOGUE character is found, a runtime error is reported.

If no codepage is specified, the external file is coded in database coding.

Example:

```
SPOOL suppliers FROM suppliersfile

SPOOL suppliers FROM /usr/transb/data/suppliersfile LOCAL

SPOOL INTO suppliers_bak SELECT * FROM suppliers
```

4.1.2 FILE Tables

```
CreateFileStatement :==
   CREATE FILE ( FileName [CodePageSpec] [NullDelimSpec])
     TABLE TableName WITHOUT IKACCESS
     ( FieldDefinition [ , FieldDefinition ] ... )

FieldDefinition ::=
    FieldName  DataTypeSpec

DataTypeSpec ::=
    DataType  |  DomainName
```

Data stored in *SpoolFiles* or other compatible file formats may be integrated into the database schema as virtual tables. These `FILE` tables offer read-only access to those files via SQL commands. They can be used throughout SQL `SELECT` statements like any other base relation.

The table definition supplies a mapping of columns in the external file to column names and Transbase datatypes.

Currently a File table can only be created `WITHOUT IKACCESS` and no key specifications are allowed. Therefore the creation of secondary indexes is currently not possible. These restrictions might be dropped in future Transbase versions.

For details on the optional parameters *CodePageSpec* and *NullDelimSpec* please consult the *Table Spool Statement*.

FILE tables are primary designed as an advanced instrument for bulk loading data into Transbase and applying arbitrary SQL transformations at the same time.

Example:

```
CREATE FILE (/usr/temp/data_file)
  TABLE file_table WITHOUT IKACCESS
  (a INTEGER, b CHAR(*))
```

```
SELECT a+10, upper(b) from file_table
SELECT b FROM file_table, regular_table
  WHERE file_table.a=regular_table.a
```

4.1.3 External File Format

For building up a Transbase database from given text files, the *DelimSpec* and the *NullSpec* are of importance for scanning the text files. With the *DelimSpec* the separator between 2 field values in the text file can be specified (the default value is the tabulator). With the *NullSpec* the textual encoding of a SQL NULL Value is specified (the default value is a question mark `?`).

If not explicitly stated differently, the following description of tuples in text files both applies to the format generated by the spooler and to the format accepted by the spooler:

- Each line of text corresponds to one tuple.
- By default, fields are separated by one or more tabulators (TAB) unless differently specified by the *DelimSpec*. The *DelimSpec* always is exactly one character.
- By default, the character `?` represents a null value of any type unless differently specified by the *Nullspec*. The *NullSpec* always is exactly one character.
- The representation of INTEGER, REAL, NUMERIC, BOOL, DATETIME and TIMESPAN values corresponds to those of *IntegerLiteral*, *RealLiteral*, *NumericLiteral*, *BoolLiteral*, *DatetimeLiteral* and *TimespanLiteral* in the TB/SQL syntax. Integers, reals, numerics and timespans can be preceded by an `-` sign.

For text strings, the following rules apply:

- The empty string is represented as a sequence of two single quotes.
- A non-empty string $x_1 \cdots x_n$ is spooled out with single quotes and as sequence of transformed characters $'\sigma(x_1) \cdots \sigma(x_n)'$. In most cases $\sigma(x_i) = x_i$ holds. However, characters which have a special meaning must be escaped. Thus, for some characters x, $\sigma(x)$ is a two-character-sequence of a backslash (’
  ’) and the character x. Special characters and their representation are shown in the table below.

As input for the spooler, the string can be represented as $x_1 \cdots x_n$ as well as $'x_1 \cdots x_n'$ (i.e. the spooler eliminates surrounding quotes).

Special characters and their representation inside strings are shown in table 4.1.

Special Character	Representation
'	\'
⟨*tab*⟩	\t
⟨*newline*⟩	\n
\	\\

Table 4.1: Special Characters in Spool Files

Special Rule for *Binchar*: As stated above, when spooling tables from external files, the spooler accepts strings in the form xyz as well as 'xyz', although the form xyz is not a valid SQL literal for the type (VAR)CHAR(p) or CHAR(*). This is comfortable for data migration into Transbase but has the consequence that table spooling compromises type compatibility (as described in chapter 2.4) in the case of CHAR and BINCHAR. Inside a spool file, values for a BINCHAR field must be written as *BincharLiteral*s, i.e. in the form 0xa0b1c2 etc. as described in Chapter 2.7.4. Whereas a value in the form xyz is accepted for a CHAR field, the same value is not accepted for a BINCHAR field because special values in that form would not be parsable in a unique manner, e.g. 0xa0b1c2 could be interpreted as a 8 byte CHAR value or a 3 byte BINCHAR value.

4.1.4 Key Collisions

When a table is spooled then Transbase rejects the data if there are 2 different tuples with the same key. In this situation the data to be spooled is inconsistent with the table creation specification. It may be advantageous to use Transbase to find out all tuples which produce a key collision. For this, recreate the table with the desired key but extended to all fields. For example, if the table T has the key on k1 and other fields f2,f3,f4, then create a table TFK with the clause: KEY IS k1,f2,f3,f4.

Then spool the table which in any case now works (except there are syntactical errors in the spoolfile). To find out all tuples with the same key, issue the query:

```
SELECT *
FROM TFK
WHERE k1 IN
(  SELECT k1
   FROM TFK
   GROUP BY k1
   HAVING COUNT(*) > 1
)
ORDER BY k1
```

4.1.5 Spooling BLOB objects

For a table which has one or several BLOB fields, the corresponding data in a spool file does not contain the BLOB objects themselves but contains file names instead. Each BLOB object is represented by a file name and the BLOB object itself is stored in that file.

This holds for input spool files as well as for output spool files produced by Transbase.

4.1.5.1 Spooling a table with BLOBs from a file

Assume a 3-ary table "graphik" with field types CHAR(20), INTEGER and BLOB.

The following statement would spool data from a file spoolfile into the table:

```
SPOOL graphik FROM spoolfile
```

The file spoolfile may look like:

```
'image31' 123    BLOBFILE001
'image32' 321    BLOBFILE002
'image33' 987    BLOBFILE003
```

The file could also contain absolute path names and then would look like:

```
'image31' 123    /usr/tmp/BLOBFILE001
'image32' 321    /usr/tmp/BLOBFILE002
'image33' 987    /usr/tmp/BLOBFILE003
```

4.1.5.2 Spooling a file from a table with BLOBs

When a file is spooled from a table with BLOBs, Transbase creates a subdirectory the name of which is that of the spoolfile prefixed with `b_` and places all BLOBs in files in that subdirectory. The names of the files are `B0000001` etc. with increasing numbers.

Assume again the table graphik described above and the following statement:

```
SPOOL INTO spfile SELECT * FROM graphik
```

The created file spfile would look like:

```
'image31' 123    b_spfile/B0000001
'image32' 321    b_spfile/B0000002
'image33' 987    b_spfile/B0000003
```

4.1.6 Filename Adaption on non-UNIX Operating Systems

If the application and/or the server is running on a non-UNIX operating system, the filename syntax requires some consideration. In the following, the filename translation mechanisms that Transbase uses are described.

In Transbase SQL, filenames occur in 3 different places: in the *SpoolTableStatement* as specification of the source file, in the *SpoolFileStatement* as specification of the target file and inside spoolfiles as BLOB placeholders.

On all three places mentioned above, Transbase SQL allows filenames in UNIX syntax as described in the preceeding chapters. This means that all examples about data spooling and BLOB filenames in spoolfiles also would be legal when the application and/or the databse server run on VMS or MS WINDOWS.

When communicationg with the operating system, Transbase translates the filenames into valid system syntax. The '/' character is thereby interpreted as the delimiter between different directory levels.

For example, on a VMS machine the UNIX filename `/usr/tmp/BLOBFILE003` would be translated into a VMS filename `[usr.tmp]BLOBFILE003`. The same name would be mapped onto `\usr\tmp\BLOBFILE003` in a WINDOWS environment.

It is also legal to use VMS or WINDOWS filename syntax if the application is running on VMS or WINDOWS, resp. For example, the statement

```
SPOOL graphik FROM [usr.tmp]graphikspf
```

would be legal on a VMS client.

A (slight) restriction, however, is that inside each part of a VMS filename, the character '/' is not allowed because it would be interpreted by Transbase as a delimiter for a composed filename. For example a file with the pathological basename `grap/hikspf` cannot be used for Transbase on VMS.

Also note that Transbase maps UNIX-like filenames to VMS or WINDOWS-like style but not vice versa. If portability is required for applications and/or spoolfiles with BLOBs, filenames should be written in UNIX syntax.

4.2 The XML Data Spooler

4.2.1 Introduction to XML

4.2.1.1 General Characteristics of XML

The eXtensible Markup Language (XML) is used to represent structural data within text documents. An XML document consists of nested elements describing

the document structure. Nesting means, that each element may have one or more child elements, each containing further elements. The application data is stored within attributes or the content of elements. The usage of tags containing the element names makes XML data self-describing. Since an XML document may have only a single root element, the hierarchical structure of an XML document can be modeled as a tree, also known as document tree. In Fig. 4.1, a small XML document and its document tree is shown.

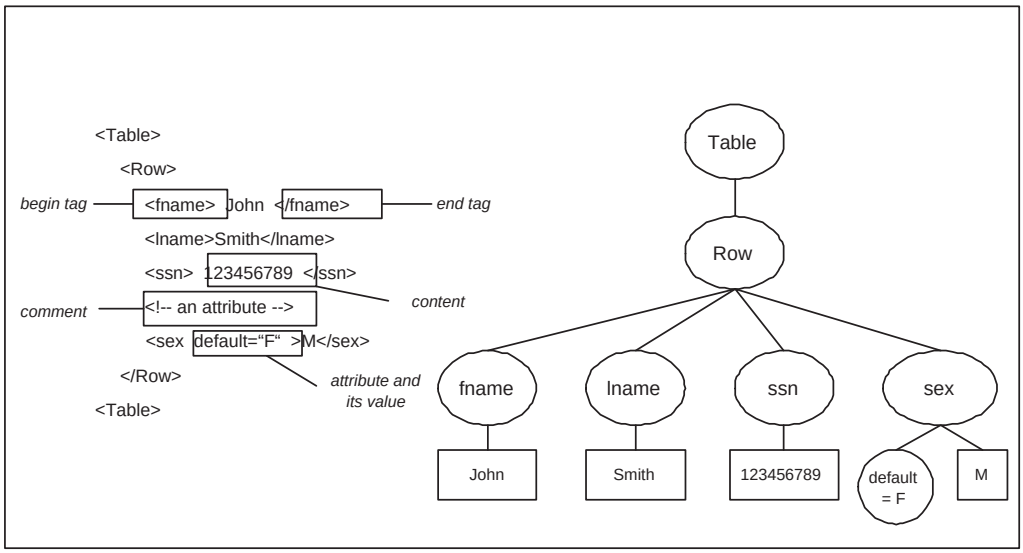

Figure 4.1: Example of an XML Document and the Document Tree

An XML document also may contain comments that do not belong to the data nor to the structure. Comments begin with '$\langle ! - -$' and end with '$- - \rangle$' (see Fig. 4.1).

The context of XML elements must not contain the signs $\rangle$, $\langle$, &, ", and '. They have to be presented as escape symbols. Likewise, escape symbols have to be used for the german signs ä, ö, ü, and ß. In Table 4.2, the special characters and their escape symbols are shown.

4.2.1.2 The Syntax of the XML Spool File

The XML spooler works only with XML documents that have a restricted structure. An XML spool file may only have four levels: the root level, whose tag is labeled with *Table*. The next level serves as delimiter between tuples and its tags are named *Row*. The third level consists of XML elements displaying the fields of the tuple. There are two possibilities to present elements of the third level:

character	symbol
>	<
<	>
&	&
"	"
'	'
Ä	Ä
Ö	Ö
Ü	Ü
ä	ä
ö	ö
ü	ü
ß	ß

Table 4.2: Special Characters

1. The names of the elements have to be identical to the column labels of the destination table.
2. The elements are labeled with *Field* and have to carry an attribute *name* whose value displays the name of the table column.

Example:

```
<lname>Smith</lname>
<Field name="lname">Smith</Field>
```

These two line both have the same meaning.

Finally, values are presented as content of XML elements at the fourth level. The XML elements also may carry attributes. At the first level, the attributes *name* and *nullrep* are defined. The first defines the name of the destination table. The later one is used for the definition of the null representation. Its meaning is explained in section 4.2.3.3. For the second level (i.e. Row), only the attribute *name* is known by the spooler. The attributes defined for the third level and their meanings are declared in section 4.2.3.5.

An example document with the document tree containing the four levels is shown in Fig. 4.1.

According to this syntax rules, there are tag names with special meaning in the spool file called delimiter tags here after wards: *Table*, *Row*, *Field*, and *Column* (see section 4.2.3.2). The XML spooler is case insensitive concerning these labels, i.e. one can write ROW or row and so on.

In contrast to this, in the DSV data spooler (delimiter separated values mode - see also section 4.1.3), the values of each tuple are presented in one line in the spool file. Usually, the values are separated by a tabulator ('\t') sign. One tuple must have the same number of elements as there are columns in the destination table. Furthermore, these elements need to have the same ordering as the table columns. NULL values are usually presented as '?' per default.

Example: In Fig. 4.2, a spool file suited for the spooler in the DSV mode is shown. It is used to transfer data into the table *supplier*. The CREATE statement of that table is defined as follows:

```
CREATE TABLE supplier(supno INTEGER NOT NULL,
                      name VARCHAR(*),
                      address VARCHAR(*),
                      PRIMARY KEY(supno))


5   DEFECTO PARTS       16 BUM ST., BROKEN HAND WY
52  VESUVIUS, INC.      512 ANCIENT BLVD., POMPEII NY
53  ATLANTIS CO.        8 OCEAN AVE., WASHINGTON DC
```

Figure 4.2: DSV spool File

The spool file shown in Fig. 4.2 has three tuples. In the XML spool file, additionally the structural information, as described above, is required.

Fig. 4.3 shows an XML spool file containing the same data as shown in the DSV spool file from Fig. 4.2.

In contrast to the DSV spool file, within an XML spool file the order of the tuple fields does not matter. Furthermore, additional elements may be present or elements can be missing (see also section 4.2.2.1). This provides more flexibility in order to transfer query results into a database table which scheme does not exactly match the output of the query.

Note: The Transbase XML spooler is not able to read XML documents containing a Document Type Description nor is it able to manage documents with namespace declarations.

```
<Table>
   <Row>
      <Field name="supno">5</Field>
      <Field name="name">DEFECTO PARTS</Field>
      <Field name="address">16 BUM ST., BROKEN HAND WY</Field>
   </Row>
   <Row>
      <Field name="supno">52</Field>
      <Field name="name">VESUVIUS, INC.</Field>
      <Field name="address">512 ANCIENT BLVD., POMPEII NY</Field>
   </Row>
   <Row>
      <Field name="supno">53</Field>
      <Field name="name">ATLANTIS CO.</Field>
      <Field name="address">8 OCEAN AVE., WASHINGTON DC</Field>
   </Row>
</Table>
```

Figure 4.3: XML Spool File

4.2.2 Principal Functionality of the XML Spooler

4.2.2.1 Tranfering XML Data Into the Database

Syntax:

```
SPOOL <tablename> FROM <file> [FORMAT XML|DSV] [NULL [IS]
StringLiteral]
```

Explanation: ⟨tablename⟩ is the name of the destination table where the tuples of the spool file are inserted. ⟨file⟩ presents the file name of the spool file. DSV stands for delimiter separated values. If the statement contains no format option the DSV mode is used per default. XML signals the XML spooling mode. With the 'NULL IS' option, the null representation can be defined (see also section 4.2.3.3).

In case of the XML mode, the spooler scans the provided spool file and reads until the end of a tuple is reached (signaled by the end tag ⟨/Row⟩). If a field is missing, the default value is inserted in the database. If no default value is available, the NULL value is used for that field. If there are additional fields for which no column in the destination table can be found, these fields are ignored.

So for example, the spool file of Fig. 4.4 contains one tuple for the table *supplier* (see section 4.2.1.2). The ordering of the fields does not match with the ordering

```
<Table>
   <Row>
      <address>64 TRANQUILITY PLACE, APOLLO MN</address>
      <anything>?</anything>
      <supno>57</supno>
   </Row>
</Table>
```

Figure 4.4: Complex XML spool File

of the table columns. The field 'name' is missing and since no default value is present, this field gets the value NULL. Furthermore, the tuple of the spool file contains a field labeled 'anything' which is ignored because the table *supplier* does not have any column of that name.

4.2.2.2 Extracting Query Results Into an XML Document

Syntax:

```
SPOOL INTO <file> [FORMAT XML|DSV] [NULL [IS] StringLiteral]
select_statement
```

Explanation: ⟨file⟩ presents the name of the output spool file. If the format option XML is used the result of the entered statement is formatted to XML. The *select statement* can be any valid select statement.

As explained in section 4.2.1.2, the root of an XML spool file is labeled *Table*. The information of each tuple is presented within the begining and ending *Row* tag. For each tuple field, the name of the associated column is presented as attribute of the beginning *Field* tag. Between the beginning and the ending *Field* tags, the query result for this field is printed (see Fig. 4.3).

4.2.3 Extended Functionality of the XML Spooler

4.2.3.1 Reading the XML Declaration

XML documents optionally may have an XML declaration which always is located at the beginning of the document. Among the version number, this declaration may include information about the encoding. The latter one may be of interest for the XML spooler.

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

The XML Spooler notices only the value of the encoding attribute within the declaration. All other information is ignored. However, at the moment, the XML spooler supports only UTF-8 encoded XML documents.

4.2.3.2 The Usage of Format Information

The XML spooler provides the opportunity to add format information as a header in front of the tuples. Such information are declared for the datetime and timespan types, so far. They are used to specify the range of these types for the complete spool file. Nevertheless, another format description may be entered as attributes within a tuple field. Furthermore, within the format information, the null representation and the default value of a table column may be defined. The format information has to be placed after the root tag (i.e. before the first tuple). For each column of the destination table, which requires additional information, an XML element named *Column* carrying several attributes is defined. This kind of information is called format information header here after wards.

```
<Table>
   <column name="bdate" type="datetime[yy:dd]" nullrep="x"/>
   <column name="age" type="timespan[dd:ss]" default="22592 12:2:4"/>
   <Row>
        ...
   </Row>
</Table>
```

Figure 4.5: Format Information Header

Example: Fig. 4.5 shows an example of the format information header at the beginning of the spool file. For the *Column* element the attributes *name*, *type*, *nullrep*, and *default* are defined. With the value of the *name* attribute, the column of the destination table is identified. Accordingly, the *type* attribute carries the type definition and the range specification of this column as value. If the *nullrep* and/or the *default* attributes are present they define the representation of the null and/or the default value for the according table column. Because of the format information shown in Fig. 4.5, the XML Spooler supposes that values of the column *bdate* for example are of type datetime and are formatted beginning with the year and ending with days. Accordingly, values of the column *age* are of type timespan, beginning with days and ending with seconds. If this field is missing in one of the tuple, the default value '22592 12:2:4' is inserted.

The meaning of the *nullrep* attribute is explained in section 4.2.3.3. Morover, the usage of the *default* attribute is explained in section 4.2.3.4.

The Usage of Format Information when Transferring Data Into the Database Together with the option explained above, there are four possibilities how the XML spooler determines which format should be used for a tuple field:

1. Datatime or timespan values can be represented in the TB/SQL syntax, i.e. the format information is written in front of the data (see section 4.1.3).

   Example:

   ```
   <Table>
      <Row>
         ...
         <age>timespan[dd:ms](19273 12:2:4:1)</age>
      </Row>
   </Table>
   ```

2. The type and range specification is declared by an XML attribute. According to this, the XML parser listens for the attribute named *type* within the beginning tag of a tuple field.

   Example:

   ```
   <Table>
      <Row>
         ...
         <bdate type="datetime[yy:dd]">1945-12-8</bdate>
      </Row>
   </Table>
   ```

   If the parser determines this attribute, it remembers its value until it can be used for type checking before inserting the tuple in the database.

   Note: There is also the possibility to enter format information as TB/SQL syntax and additionally provide the concerning XML attributes. In this case, the attributes are ignored.

Example:

```
<today type="datetime[yy:dd]">
   datetime[yy:hh](2007-12-11 15)
</today>
```

In this case, the spooler assumes that the range specification of [yy:hh] is correct.

3. A header containing the format information as described above may be present. This information is only used, if the parser did not find such information within the field declaration (either as XML attributes or as TB/SQL representation).

4. If there is neither any format information within the field declaration nor any format information header present, the XML spooler assumes that the appropriate value has the format as defined in the column of the destination table.

Note: If the format of the value does not match the format to be used according to the added format information or the database scheme, an error handling is started (see section 4.2.4).

Writing the Format Information for Query Results If a query result contains fields of type timespan or datetime, a header containing the format information as described above is generated and written into the output spool file.

Example:

```
<Table>
   <column name="bdate" type="datetime[yy:dd]"/>
   <column name="age" type="timespan[dd:ss]"/>
   <Row>
      ...
  </Row>
</Table>
```

4.2.3.3 The Representation of Null Values

With the XML spooler, there are several opportunities, to declare the representation of the null value: the definition of a single character within the spool

statement, to add a null representation string as attribute of the *Table* element, or to use the *nullrep* attribute within the *Column* element. If none of these three possibilities is used, the default ('?') is used for the representation of the null value.

Table Spool Statement with a Null Representation The table spool statement provides the option to enter a single character representing the Null value (see section 4.1.1).

Example:

```
spool employee from test.xml format xml null is 'x'
```

If an 'x' is scanned for a field value, the spooler generates a NULL to be inserted in the database. If the value 'x' should be inserted instead of NULL, in the spool file the attribute *null* has to be set to false.

Example:

```
<Table>
   <Row>
      <Field name="lname" null="false">x</Field>
      ...
   </Row>
   ...
</Table>
```

Note: The XML spooler also supports the representation of null values by setting the *null* attribute of the *Field* element to true. Hence, the following two lines have the same meaning, if the null representation is set to 'x':

```
<Field name="lname">x</Field>
<Field name="lname" null="true"/>
```

The Null Representation for the Complete Document As mentioned in section 4.2.1.2, the *Table* element may have an attribute named *nullrep*. Its value displays the representation of the null value for the remaining document. In contrast to the representation of the table spool statement, this value may be a string, not only a single character. If the *nullrep* attribute is present within the *Table* tag, the null representation of the spool statement - if any - is ignored. Again, if for a tuple field the same value as for the null representation should be inserted in the database, the *null* attribute has to be set to false.

Example:

```
<Table nullrep="xyz">
   <Row>
      <Field name="lname">x</Field>
      <Field name="rname" null="false>xyz</Field>
      <Field name="address">xyz</Field>
      ...
   </Row>
</Table>
```

Since the *nullrep* attribute is present, the value 'x' for the field *lname* is not interpreted as null although it was defined as null representation by the spool table statement. Thus, the following tuple values are inserted in the database:
x, xyz, NULL, ...

The Null Representation for a Single Column Within the format information header described in section 4.2.3.2, it is possible, to declare a value for the null representation. This is done with the *nullrep* attribute within the *column* element. As for the null representation of the *Table* element, the value may be a string. If this attribute is entered there, the value defines the null representation only for the column of the specified name. Other null representations (that from the *Table* element or that of the spool statement) then are not applied to the specified column. Again, if a value to be inserted in the database is the same as the null representation value, the attribute *null* has to be set to false.

Example:

```
<Table nullrep="xyz">
   <Column name="lname" nullrep="abc"/>
   <Row>
      <Field name="lname">abc</Field>
      <Field name="rname">xyz</Field>
      ...
   </Row>
   <Row>
      <Field name="lname" null="false">abc</Field>
      <Field name="rname" null="false">xyz</Field>
      ...
   </Row>
    <Row>
      <Field name="lname">xyz</Field>
      <Field name="rname">x</Field>
```

```
      ...
   </Row>
 </Table>
```

Although, if in the spool statement the NULL IS 'x' option was used, the following tuple values are generated and inserted in the database:

```
NULL, NULL, ...
abc,  xyz,  ...
xyz,  x,    ...
```

The Default Value for the Null Representation If no null representation is present (neither in the spool statement nor nor in the spool file), the default null symbol ('?') is used. This is also true for the DSV Spooler. Also in this case, it is necessary, to set the field attribute *null* to false if the value ? has to be inserted in the database.

Writing the Null Representation When writing the query result in an XML document, the *Table* element gets the attribute *nullrep* in any case. At the moment, the value of this attribute can be only a single character. The value is either the default null symbol ('?') or was entered with the file spool statement. Furthermore, it is not possible to define a null representation for a single column.

Example:

```
spool into test.xml format xml null is 'x' select * from employee
```

In this case, the output document looks as follows:

```
<Table nullrep="x">
   <Row>
      <Field ....
   </Row>
   ...
</Table>
```

4.2.3.4 The Default Values

After a complete tuple was scanned by the XML spooler, for fields that are not present the default value if any available is inserted in the data base. Otherwise, for these fields a NULL is inserted. There are two possibilities to define the default values: first, default values can come from the table description. Second, within the format information header, an attribute declaration can be used to define the default value. These possibilities are explained next.

Default Values from the Table Description Default values that come from the table description are declared within the CREATE TABLE statement or with an ALTER TABLE statement.

Example: In the following example, a spool file is used to transfer data in a table called *employee*. The CREATE TABLE statement of this destination table looks as follows:

```
CREATE TABLE employee (..., fname VARCHAR(*) DEFAULT 'MARY', ...)
```

For each tuple, where the field *fname* is not present, the value "Mary" is inserted.

If the default value of the table description represents a sequence, this sequence has to be updated each time the default value is used.

Example: In the following, parts of a spool file are shown that should be transfered into a data base table:

```
<Table>
  <Row>
     <Field name="ssn">20</Field>
     ...
  </Row>
  ...
</Table>
```

The field 'ssn' is of type integer and has as default a sequence. In the first tuple, this field is present. In all other tuples not shown, the field is missing and hence, the sequence is increased each time the default value is inserted.

Note: If there are more than one sequences per table, all sequences are increased at the same time. Hence, more sequences may result in confusing values.

Default Values within the Format Information Header In order to declare a default value within the format information header, the attribute named *default* is used.

Example:

```
<column name="fname" default="Max"/>
```

In this case, for missing fields with the name *fname*, the default value "Max" is inserted.

If the attribute *default* is present within the format information header, the XML spooler checks if its value is valid according to the type declaration of the table description. If an uncorrect value was entered the definition of the default value is ignored.

Note: The definition of the default value within the format information header has a higher priority than that of the table description. I.e. if both, the table description and the format information header contain a default value for the same field, the default value of the table description is ignored.

4.2.3.5 XML Attributes Known by the XML Spooler

Attribute Name	Possible Values
name	any string
nullrep	any string
type	datetime[cc:cc] | timespan[cc:cc]
null	true|false
default	any string
blobfile	any string
offset	any number
blobsize	any number
encoding	any string

Table 4.3: Attributes and Their Values

List of Attributes Table 4.3 shows the attributes known by the XML spooler and their possible values. If the spool file contains other attributes as declared within Table 4.3, these attributes are ignored by the spooler. Similarly, if the parser encounters a not expected attribute value, depending on the location, an error is generated as explained in chapter 4.2.4.

Attributes Describing Format Information As described in section 4.2.3.2, the parser of the XML spooler has to know the following attributes within a *Column* element: *name*, *type*, and *default*. The attributes *name* and *type* are also known within the beginning tag of a tuple field.

The Attributes for Null Values As explained in section 4.2.3.3, for both - the DSV and the XML spooler, the default null symbol is presented by the '?' sign. Furthermore, a single character for the null representation may be entered with

the spool statement. Within an XML spool file, the attribute labeled with *nullrep* may be used to overwrite this null representation for the complete document or only for a single column (see section 4.2.3.3). Additionally, the attribute 'null' can be used to signal the usage of a null value for a particular field. If this attribute carries the value 'true', a NULL is inserted for the appropriate tuple field. There are three possibilities, to declare a NULL field with this attribute:

1. The null attribute is set to true and no value is entered. In this case, usually no closing tag is required because the opening tag is closed after the attribute declaration.

   Example:

   ```
   <today null="true"/>
   ```

2. Although no value is entered, it is valid to use the opening and closing tag within the XML document.

   Example:

   ```
   <today null="true"></today>
   ```

3. The null field may carry a value.

   Example:

   ```
   <today null="true">2007-12-07</today>
   ```

   Since the attribute value null is set to true, the data entered between opening and closing tag is ignored and a NULL is inserted for this tuple field.

Attributes Defining Blob Characteristics The attributes *blobfile*, *offset*, and *blobsize* are used when spooling blob. More details for these attributes can be found in chapter 4.2.5.

Attributes of the XML Declaration As already explained in section 4.2.3.1, an XML document optionally may have an XML declaration including attributes. The parser only remembers the value of the 'encoding' attribute, all other attributes within this declaration are ignored.

4.2.4 Error Reports

The transbase error handling differs between hard and weak errors. If an hard error occurs, the insertion process is stopped immediately and a roll back is performed. Hence, in case of an hard error, no tuple from the spool file is inserted. If a weak error is encountered, the appropriate tuple is ignored and skipped and all correct tuples are inserted.

4.2.4.1 Hard Errors

Concerning the XML spool mode, hard errors occur in connection with blobs or if an unexpected tag is encountered. I.e., if at least one column of the destination table is of type blob, each encountered error is handled as an hard error. The error of an unexpected tag occurs especially in connection with the delimiter tags defined in section 4.2.1.2. So for example, an XML spool file my begin only with an XML declaration or with the *Table* tag. After the beginning *Table* tag, the XML spooler accepts only a *Column* or a *Row* Tag. At the end of a *Column* tag, a further *Column* tag or a beginning *Row* tag is required. Finally, after a closing *Row* tag, only a beginning *Row* or an ending *Table* tag is allowed. If the spooler encounters another tag as expected, the spool process is aborted since no realistic recovery point is available.

4.2.4.2 Weak Errors

XML Syntax Errors According to the error treating, there are three classes of XML syntax errors: hard errors as unexpected tags, syntax errors forcing the skipping of a tuple, and syntax errors leading in a scanning to the next recovery point. The first error class is already explained in section 4.2.4.1, the other two classes will be explained next.

1. XML Syntax Errors Leading in a Skip Operation:
   If an XML syntax error occurs that still allows the correct interpretation of the following XML segments (i.e. tag, attribute, value, ...), the rest of the tuple is skipped. This means, the tuple is not inserted into the database but is written in the error report with an appropriate error message as XML comment.

   Example: The end tag differs from the beginning tag as shown next.

```
<Row>
   <fname>John</fname>
   <lname>Smith</lname>
```

```
   <ssn>123456789</ssn>
   <address>731 Fondren, Houston, TX</add>
   <sex>M</sex>
</Row>
```

In the example, the fourth tuple field starts with an *address* tag and ends with an *add* tag. In this case, the complete tuple is written in the error file that contains all incorrect tuples along with the proper error message. The spooling then starts at the beginning of the next tuple.

Error Message:

```
<Row>
   <fname>John</fname>
   <lname>Smith</lname>
   <ssn>123456789</ssn>
   <!-- missmatch between open and closing tag -->
   <address>731 Fondren, Houston, TX</add>
   <sex>M</sex>
</Row>
```

2. XML Syntax Errors Leading in an Ignore Operation:
   If in the XML spool file an unexpected sign occurs, the spooler is not able to interpret the meaning of the following segments. Hence, nothing of the tuple is inserted in the database. The spooler ignores everything until to the next recovery point. A recovery point can be found at the beginning of each tuple, i.e. if a beginning *Row* tag is encountered.

   Such errors are for example missing angles, forgotten inverted commas, and so forth. Due to the restricted syntax of XML spool files, the transbase spooler also interprets mixed content as syntax error.

   Example:

```
<Row name="row1">
   <field name="fname">?</field>
   this text is mixed content
   <lname>?</lname>
   ...
</Row>
<Row name="row2"'>
   ...
</Row>
```

After a closing *Field* tag, only an oping *Field*, an closing *Row* tag, or a XML comment is expected. When scanning the text after the first tuple field, the spooler ignores the rest of the tuple and starts the spooling process at the begin of the next tuple (⟨Row name="row2"⟩). In the error report, the following error message is written.

```
<Row name="row1">
   <field name="fname">?</field>
   <!-- XML Syntax error found, scanning to begin of next tuple  -->
</Row>
```

Errors Occurring During the Spooling of a Tuple There are several errors that may occur during the parsing process of the tuple. If such an error is determined, the rest of the tuple is skipped. The incorrect tuple is written in an error report file where an error message is inserted before the faulty tuple field. Especially wrong types, invalid null definitions, or invalid attribute values may occur during the tuple spooling. These errors are explained next.

1. Invalid Null Definition:
   If in the table description a field is declared to be not null and in the spool file for this field the null attribute is set to true or the value for the null representation was entered, this tuple is skipped. In the following example, the field address must not be null according to the table description.

   Example:

   ```
   <Row>
      <fname>Franklin</fname>
      <lname>Wong</lname>
      <ssn>333445555</ssn>
      <address null="true"/>
      <sex>M</sex>
   </Row>
   ```

   In the error file the following error message is entered:

   Error Message:

   ```
   <Row>
      <fname>Franklin</fname>
      <lname>Wong</lname>
      <ssn>333445555</ssn>
      <!-- field must not be null -->
      <address null="true"/>
      <sex>M</sex>
   </Row>
   ```

2. Wrong Types:
   Such errors occur for example, if a string is added where a numeric value is supposed.

   Example:

```
<Row>
   <fname>Joyce</fname>
   <lname>English</lname>
   <ssn>453453453</ssn>
   <address>563 Rice, Houston, TX</address>
   <sex>M</sex>
   <salary>error</salary>
</Row>
```

   In the example above, the field salary is of type numeric. During the tuple spool, a string value is scanned and hence the error handling is started.

   Error Message:

```
<Row>
   <fname>Joyce</fname>
   <lname>English</lname>
   <ssn>453453453</ssn>
   <address>563 Rice, Houston, TX</address>
   <sex>M</sex>
   <!-- numeric error -->
   <salary>error</salary>
</Row>
```

3. Errors Concerning XML Attributes:
   In Tab. 4.3, the attributes and their possible values are listed. Errors concerning XML attributes are encountered if a not expected value was entered.

   The attributes can be classified in three categories: attributes describing format information, attributes describing characteristics of blobs, and attributes belonging to the *Field* element. The error handling for attributes depends on this classification.

   (a) Errors Within Field Attributes:
   Usually, errors are encountered during the spooling stage which causes the skipping of the tuple and the generation of an appropriate error message. An example of an incorrect attribute value and the according error message is shown below.

Example:

```
<Row>
   <fname>James</fname>
   <lname>Borg</lname>
   <ssn>888665555</ssn>
   <bdate null="nonsense">1975-11-30<bdate>
   <sex>M</sex>
</Row>
```

In this case, the attribute 'null' of the XML element 'bdate' has the value 'nonsense'. Since for this attribute only the values 'true' or 'false' are defined, the following error message is generated.

```
<Row>
   <fname>James</fname>
   <lname>Borg</lname>
   <ssn>888665555</ssn>
   <!-- not expected attribute value (nonsense)  -->
   <bdate null="nonsense">1975-11-30<bdate>
</Row>
```

(b) Errors at Attributes Describing Blob Functionality:
As mentioned above, if the destination table of the spool statement contains at least one blob column, each error is classified to be an hard error. Due to this, wrong values for the attributes *offset*, and *blobsize* result in an hard error. Hence, in such a case, no tuple of the spool file is inserted in the database.

(c) Errors at Attributes Describing Format Information:
As described in section 4.2.3.2, attributes that describe format information may occur in the format information header or within the begin tag of the tuple field.

The correctness of the type and range specifications is verified at the insertion stage when the complete value is available. If there is an error encountered, the tuple is skipped and an error message is generated. By the reason of this, if there was entered an incorrect type (especially in the range part) within the format information header this results in the skipping of all tuples that use this information in the remaining spool file.

Errors Occurring at the Insertion Stage After a tuple is scanned completely, problems may occur before or at the insertion step. So for example, which tuple fields are missing can be determined only after the complete tuple was parsed. Furthermore, integrity violations and key collisions can be recognized only when trying to insert the tuple. If such an error occurs, the tuple is not

inserted in the database. It is written in the error file together with a proper error message. Since the error concerns the complete tuple and not only a singe tuple field, the error message is placed in front of the tuple declaration. The spooling process goes ahead with spooling of the next tuple. An example of such an error is explained below.

Example: As explained in section 4.2.1.2, it is not necessary to declare all tuple fields within an XML spool file. For a missing field the default value is inserted. If no default value is declared the NULL value is used. However, if such a field may not be null according to the table description the tuple must not be inserted in the database and hence, the error handling is started.

```
<Row>
   <fname>James</fname>
   <lname>Borg</lname>
   <ssn>888665555</ssn>
   <sex>M</sex>
</Row>
```

In this example, the field address which was declared to be not null is not present within the tuple. Hence, if no default value is available, the following error message is generated.

Error Message

```
<!-- The field address must be declared - not null -->
<Row>
   <fname>James</fname>
   <lname>Borg</lname>
   <ssn>888665555</ssn>
   <sex>M</sex>
</Row>
```

4.2.4.3 Attempt to Use an XML Document in the DSV Spooling Mode

The spool statement allows optional to choose the spooling mode (DSV or XML). If the format part is not entered, the DSV spooler is used per default. It may happen, that the format specification was forgotten and the user attempts to spool an XML document in the DSV spooling mode. In such a case, at the end of the spooling process, a proper error message is generated (error in DSV spooler - possibly forgot to enter 'format xml' option in statement).

For this error message, two conditions have to be fulfilled:

1. The first scanned symbol may be the start of an XML document.
2. There is at least one error for each line:
   If an XML document is used with the DSV spooling mode usually no correct tuple is encountered in the spool file, i.e. there are as many errors as spooled lines.

4.2.5 Spooling of Blobs with the XML Spooler

4.2.5.1 Transferring Blobs Into the Database

In the mode of delimiter separated values, the spool file usually contains file names for each blob field. The blob data is stored in the associated files (see section 4.1.5). The spooling process is performed by two runs: in the first scan, the file names are collected and requested from the client. The client than sends these files to the server. In the second scan, the remaining values of the spool file are read and the complete tuples are inserted in the data base. There is also the option to spool several blobs stored in one file by the usage of offset and size values.

If the XML mode is used, the file names of the blobs are entered as values of the attribute *blobfile* at the according tuple fields. The spool statement is the same as explained in section 4.2.2.1. Fig. 4.6 shows an example of an XML spool file containing blob file names. It is used to spool tuples in the table *blobex* which was created with the following statement:

```
CREATE TABLE blobex (nr INTEGER NOT NULL,
                     picture BLOB,
                     PRIMARY KEY (nr))
```

4.2.5.2 Writing Blobs from a Query Result

If a query result contains blob columns, the blobs usually are written in a separate file. The output spool file then contains the name of these files. In order to this, the spool file may look like that shown in Fig. 4.6.

4.2.5.3 Inline Blobs

As in the DSV spooling mode, the blob data may also be entered as inline information. In an XML spool file, the inline blob data is presented as value between the proper opening and closing field tags. For inline blobs, the attributes *blobfile*, *blobsize*, and *offset* are not present. Hece, if none of those attributes was entered, the spooler assumes that the value between open and closing tag belongs to an inline blob. Inline blobs are only useful if the blob is not too large. Inline blobs have to be encoded with hex representation or with the base64 (for pictures).

```
<Table>
   <Row>
      <nr>1</nr>
      <picture blobfile="B0000001.001"/>
   </Row>
   <Row>
      <nr>4</nr>
      <picture blobfile="maske.jpg"/>
   </Row>
   <Row>
      <nr>6</nr>
      <picture blobfile="photo.jpg"/>
   </Row>
</Table>
```

Figure 4.6: XML Spool File Containing Blobs

Example:

```
<picture>/9j/4AAQSkZ    ...            </picture>
```

In this example, the value of the blob represents parts of the code of a jpg-encoded picture.

While in the DSV spooling mode, mixing of inline blobs and blobs data located in a file is not possible, this mechanism is allowed in the XML spooling mode. The spooler decides because of the attributes that are available or not if the blob data is located in the spool file or if it has to be loaded from a file.

4.2.5.4 Storing Several Blobs in One File

Spooling Several Blobs into One File As in the delimiter separated value mode, also in the XML mode it is possible to spool several blobs into one file.

In the following, an example statement is presented. It allows the spooling of the content from the table *blobex* containing one blob column in the file *blobexample*:

```
SPOOL INTO blobexample BLOBFILESIZE=100 mb SELECT * FROM blobex
```

Fig. 4.7 shows the output document that is generated for the statement above when using the DSV mode. For each blob optionally the file name and a the byte offset is printed. The size of the blob always has to be present (see Fig. 4.7).

```
1   'B0000001.001<0:11505>'   'M'
4   '<11505>'                  'M'
6   '<11505>'                 'M'
7   '<11505>'                 'M'
```

Figure 4.7: Output DSV Spool File

This output specifies that the first blob can be found in the file *B0000001.001* at byte offset 0 and has a size of 11,505 bytes. Since for the second and all further blobs no file name is add the same file is used. For those blobs only the size is specified. This means, the blob starts with the end of the blob before.

In the XML mode, the size and offset values are written as XML attributes (see Fig. 4.8).

```
<Table>
   <Row>
      <nr>1</nr>
      <picture offset="0" blobsize="11505" blobfile="B0000001.001"/>
      <sex>M</sex>
   </Row>
   <Row>
      <nr>4</nr>
      <picture blobsize="11505"/>
      <sex>M</sex>
   </Row>
   <Row>
      <nr>6</nr>
      <picture blobsize="11505"/>
      <sex>M</sex>
   </Row>
   <Row>
      <nr>7</nr>
      <picture blobsize="11505"/>
      <sex>M</sex>
   </Row>
   <Row name="nr4">
      <nr>8</nr>
      <picture blobfile="photo.jpg"/>
   </Row>
</Table>
```

Figure 4.8: Output XML Spool File

Spooling Several Blobs from One File In the spool file, for each blob, optionally the filename and the byte offset has to be entered. The blob size is always required. In the XML mode, this information has to be presented as shown in the

output document from Fig. 4.8. Since for the second and all further tuples no file name is present in the spool file, the spooler uses the file (*B0000001.001*) from the tuple before. Furthermore, no byte offsets are available for these tuples. Hence, the spooling of the second blob starts and the end of the first blob and so on. If there is a blob field in a tuple where only the attribute *blobfile* is present but no size and no offset, then the spooler supposes that the complete file data belongs to one blob. So for example, for the last tuple of Fig. 4.8, the complete content of the file 'photo.jpg' is loaded in the concerning blob container.

4.3 External data sources

4.3.1 Transbase D

TransbaseD offers direct and transparent read/write access to remote Transbase databases for distributed queries and data import. Please consult *TableReference* for details on how to connect to a remote Transbase site in an SQL statement.

The following example is a distributed join using two remote databases.

Example:

```
INSERT INTO T
SELECT  q.partno, supp.sno
FROM quotations@db7@server3 q, suppliers@db9@server5 supp
WHERE q.suppno = supp.sno
```

4.3.2 JDBCReader

Additionally it is possible to transfer data from arbitrary JDBC or other database data sources via *TableFunction*s. These functions may be used throughout SQL SELECT statements like any other base relation and can be used for querying remote data, data loading and data transformation.

The JDBCREader can be used for querying remote JDBC data sources or for data import.

Example:

```
INSERT INTO T SELECT * FROM
  FUNCTION JDBCReader('conn_string','user','passwd',
  'SELECT * FROM jdbc_table')
```

Refer to *TableFunction* for more details on the configuration of the JDBCReader.

4.3.3 OraReader

Similar to the JDBCReader, the OraReader is a *TableFunction* that provides read-only access to remote Oracle databases. For maximum efficiency, the function is implemented via a dynamic link library (in C programming language) using the OCI interface to access Oracle. Thus it will outperform the JDBCReader on Oracle data sources. The function may be used throughout SQL SELECT statements just like any other base relation.

The OraReader can be used for querying remote Oracle data sources for data import.

Example:

```
INSERT INTO T SELECT * FROM
  FUNCTION OraReader('//orasrv/oradb','scott','tiger',
  'SELECT * FROM ora_table')
```

4.3.4 FILE Tables

Data stored in files may be integrated into the database schema as virtual tables. These FILE tables offer read-only access to those files via SQL commands. They can be used throughout SQL SELECT statements like any other base relation.

Example:

```
CREATE FILE ('/usr/temp/data.csv')
  TABLE file_table WITHOUT IKACCESS
  (a INTEGER, b CHAR(*))

SELECT a+10, upper(b) from file_table
SELECT b FROM file_table, regular_table
  WHERE file_table.a=regular_table.a
```

FILE tables are primarily designed as an advanced instrument for bulk loading data into Transbase and applying arbitrary SQL transformations at the same time.

Chapter 5

Data Manipulation Language

The Data Modification Language (DML) portion of TB/SQL serves to extract data tuples from tables or views (*SelectStatement*), to delete tuples (*DeleteStatement*), to insert tuples (*InsertStatement*) and to update tuples (*UpdateStatement*). The following paragraphs describe the syntax of the DML botton up, i.e. the language description starts with basic units from which more complex units can be built finally leading to the four kinds of statements mentioned above.

5.1 FieldReference

The construct FieldReference is used to refer to a specific field of a specific table.

Syntax:

```
FieldReference ::=
    UnqualifiedField | QualifiedField
UnqualifiedField ::=
    FieldName
QualifiedField ::=
    CorrelationName.FieldName
CorrelationName ::=
    Identifier
FieldName ::=
    Identifier
```

Explanation: The FieldName is the name of a field of a table. The CorrelationName is the name of a table or a shorthand notation for a table introduced

in the FROM-clause of a SelectStatement. See *SelectExpression* and "Rules of Resolution" for more details.

The following examples show the usage of Field in a SelectStatement. The last two examples explain the use of CorrelationName in QualifiedField.

Example:

```
SELECT suppno FROM suppliers

SELECT suppliers.suppno FROM suppliers

SELECT s.suppno FROM suppliers s
```

5.2 User

The construct User serves to refer to the name of the current user.

Syntax:

```
User ::=
    USER
```

Explanation: The keyword USER can be used like a StringLiteral. Its value in a statement is the login name of the user who runs the statement. The type of the value is CHAR(*).

Example:

```
SELECT suppno FROM suppliers
WHERE name = USER
```

Example:

```
SELECT tname FROM systable, sysuser
WHERE owner = userid
AND username = USER
```

5.3 Expression

An Expression is the most general construct to calculate non-boolean values.

Syntax:

```
Expression ::=
    [Unary] Primary [ Binary [Unary] Primary ] ...
Unary ::=
    +  |  -  | BITNOT
Binary ::=
    +  |  -  |  *  |  /  |  BITAND  |  BITOR |  Strconcat
Strconcat ::=  ||
```

Explanation: For Primaries of arithmetic types all operators are legal and have the usual arithmetic meaning. Additionally, the binary '+' is also defined for character types and then has the meaning of text concatenation.

For the time types also some of the operators are defined (see Chapter 9 (The Data Types Datetime and Timespan).

The operator precedences for arithmetic types are as usual: Unary operators bind strongest. BITAND / BITOR bind stronger than '*' and '/' which in turn bind stronger than binary '+' and '-'.

The operator `||` denotes concatenation of string values and is an alternative for + for strings, see example below.

Associativity is from left to right, as usual. See chapter 14 (Precedence of Operators) for a complete list of precedences.

Note: Computation of an Expression may lead to a type exception if the result value exceeds the range of the corresponding type. See Data Type and Type exceptions. See also Null Values.

Example:

```
- 5.0
-5.0
 price * -1.02
'TB/SQL' + ' ' + 'Language'
'Dear' + title + name
'Dear'  ||  title || name
+  1.1 * (5 + 6 * 7)
```

In all but the last example, the constituting Primaries are Fields or Literals. In the last example, the second *Primary* is itself an *Expression* in parentheses.

For the operands BITOR and BITAND see Chapter 10 (The TB/SQL Datatypes BITS(p) and BITS(*)).

5.4 Primary, CAST Operator

A Primary is the building unit for Expressions.

Syntax:

```
Primary ::=
    SimplePrimary
  | CastedPrimary

Casted_Primary ::=
    SimplePrimary  CAST  DataTypeSpec
  | CAST ( SimplePrimary  AS  DataTypeSpec )

DataTypeSpec ::=
    DataType
  | DomainName
```

Explanation: The functional notation CAST(...) is the official SQL2 syntax, the postfix notation is the traditional Transbase syntax.

A CAST operator serves to adapt the result of an *SimplePrimary* to a desired data type. The specified data type must be compatible with the result type of the SimplePrimary (but see also chapter 2.6 (CASTING to/from CHAR).

If the CAST operator is used on NUMERIC, FLOAT or DOUBLE values to map them into BIGINT, INTEGER, SMALLINT or TINYINT values, truncation occurs. See the example below how to round values instead.

The function TO_CHAR(expr) is equivalent for CAST(expr as CHAR(*)).

Note: CASTing may lead to a type exception if the value to be casted exceeds the range of the target type. See the Chapters 2.2 Data Types, 2.5 Type Exceptions, 5.4 CAST Operator.

Example:

```
name CAST CHAR(30)
price CAST INTEGER
(price + 0.5) CAST INTEGER
```

The last expression shows the truncation and rounding of NUMERIC or REAL values into an INTEGER value.

5.5 SimplePrimary

A SimplePrimary is the building unit for CastedPrimaries or Expressions.

Syntax:

```
SimplePrimary ::=
      Literal
    | FieldReference
    | Parameter
    | User
    | (Expression)
    | (SubTableExpression)
    | SetFunction
    | ConditionalExpression
    | TimeExpression
    | SizeExpression
    | BlobExpression
    | StringFunction
    | SignFunction
    | ResultcountFunction
    | SequenceExpression
    | ODBC_FunctionCall
    | UserDefinedFunctionCall
Parameter ::=
      # IntegerLiteral ( DataType )
    | Questionmark
    | HostVarInd
Questionmark ::=  ?
```

Explanation: *HostVarInd* is a reference to a host variable of an Embedded SQL program (see TB/ESQL Manual) and only allowed inside an ESQL program.

Parameter is the means to specify a parameter for a stored query in an application program and is only allowed inside a TBX program (see TBX Manual). The ?-notation can be used wherever the type of the parameter can be deduced from its context (e.g. `Field = ?`).

A *SimplePrimary* can be a parenthesized *Expression* which simply models that an *Expression* in parentheses is evaluated first.

If a *SubTableExpression* is used as a *SimplePrimary*, its result must not exceed one value (i.e. one unary tuple), otherwise an error occurs at runtime (see SubTableExpression). If its result is empty, it is treated as a null value (see Null Values).

Example:

```
price
0.5
(price + 0.5)
(SELECT suppno FROM suppliers WHERE name = 'TAS')
```

5.5.1 SetFunction

A SetFunction computes one value from a set of input values or input tuples.

Syntax:

```
SetFunction ::=
      COUNT ( * )
    | DistinctFunction
    | AllFunction

DistinctFunction ::=
      { AVG | COUNT | MAX | MIN | SUM }
      ( DISTINCT Expression )

AllFunction ::=
      { AVG | MAX | MIN | SUM }
      ( [ALL] Expression )
```

Explanation: SetFunctions are typically used in the SELECT-clause or HAVING-clause of a *SelectExpression* (see SelectExpression but also Rules of Resolution). Input of a SetFunction is a set of tuples or a set of values which is defined by the semantics of *SelectExpression* (see SelectExpression).

COUNT (*) delivers the cardinality of the set of input tuples. For all other forms of a SetFunction, the input is the set of values which results from application of the *Expression* to the input tuples. If a *DistinctFunction* is specified, all duplicate values are removed before the *SetFunction* is applied. The functions compute the cardinality, the sum, the average, the minimum and the maximum value of the input value set, resp.

Functions COUNT, MIN and MAX are applicable to all data types.

Function SUM is applicable to arithmetic types, Function AVG is applicable to arithmetic types and to TIMESPAN.

The result type of COUNT is INTEGER. The result type of AVG on arithmetic types is DOUBLE. For all other cases the result type is the same as the type of

the input values. Of course, the CAST operator can be used to force explicit type adaptation.

*SetFunction*s except COUNT ignore null values in their input. If the input set is , COUNT delivers 0, all others deliver the null value (Null Values).

Note: Function SUM and AVG may lead to a type exception if the sum of the input values exceeds the range of the result type. See Data Type and Type exceptions.

Example: How many distinct parts are ordered?

```
SELECT COUNT (DISTINCT partno)
FROM quotations
WHERE qonorder > 0
```

Example: How many parts are delivered by those suppliers who deliver more than 2 parts?

```
SELECT suppno, COUNT (*)
FROM quotations
GROUP BY suppno
HAVING COUNT (*) > 2
```

Example: What is the average order for each part?

```
SELECT partno, AVG(qonorder) CAST INTEGER
FROM quotations
GROUP BY partno

SELECT partno, AVG(qonorder)
FROM quotations
GROUP BY partno
```

Note: The second notation of the previous example computes the average in DOUBLE, whereas the first notation truncates the average to INTEGER.

5.5.2 WindowFunction

While SetFunctions aggregate a set of input rows into one result row, a WindowFunction calculates one result row for every input row. Here the aggregates are calculated over a set of rows in the vicinity of the current input row.

```
WindowFunction ::= window_function_name(expr_list) OVER
  ( [partition_clause] [order_by_clause] [window_clause]])

window_function_name ::= { AVG | COUNT |
  DENSE_RANK | MAX |
  MIN | RANK | SUM }

partition_clause ::= PARTITION BY {(expr_list) | expr_list}

order_by_clause ::= ORDER BY {(expr_list) | expr_list}

window_clause ::= { ROWS | RANGE } { preceding_clause |
  BETWEEN lowerbound_clause AND upperbound_clause }

preceding_clause ::= UNBOUNDED PRECEDING |
  value_expr PRECEDING | CURRENT ROW

lowerbound_clause ::= UNBOUNDED PRECEDING |
  value_expr { PRECEDING | FOLLOWING } | CURRENT ROW
upperbound_clause ::= UNBOUNDED FOLLOWING |
  value_expr { PRECEDING | FOLLOWING } | CURRENT ROW
```

Explanation: WindowFunction are useful for calculating rankings and *running* totals or *sliding* averages. They are typically used in the SELECT clause of a *SelectExpression* (see SelectExpression. They operate on a query result set, i.e. after FROM, WHERE, GROUP BY and HAVING clauses are evaluated. First this result is partitioned according to the `partition_clause`. Then each partition is processed row by row, so every row will become the *current* row once. The aggregate for the current row is calculated OVER a set of rows (window) in this partition, as defined by the `window_clause`.

OVER() distinguishes an WindowFunction from a SetFunction.

ROWS specifies that the windows is defined between absolute boundary offsets. If ROWS is specified, there are no restrictions to the following `order_by_clause` and it is completely optional. Windows boundaries refer to row positions relative to the current row.

Example: If the limits of a ROWS window are BETWEEN CURRENT ROW AND 5 FOLLOWING, then the current row and the five following rows are within the window. Therefore this ROWS window contains at most 6 rows.

RANGE specifies that the window is defined between relative boundary offsets. If RANGE is specified with an `value_expr` boundary, the `order_by_clause` is

mandatory and must contain exactly one expression. These `value_expr` windows boundaries refer to the one field used in the `order_by_clause`.

Example: If the limits of a RANGE window are BETWEEN CURRENT ROW AND 5 FOLLOWING, then the window contains all rows whose sort field is

(1) equal or larger than the sort expression of the current row and

(2) equal or smaller than the sort expression of the current row + 5.

Therefore this RANGE window can contain any number of rows.

`value_expr` is a logical or physical offset. For a ROWS window it must be a positive INTEGER constant or an expression that evaluates to a positive INTEGER value. For a RANGE window it must be a positive constant or expression of arithmetic type or of type TIMESPAN/INTERVAL. If value_expr FOLLOWING is the start point, then the end point must be value_expr FOLLOWING. If value_expr PRECEDING is the end point, then the start point must be value_expr PRECEDING.

Defaults: If the partition_clause is missing, the defaults is PARTITION BY NULL, i.e. no partitioning us applied.

If the order_by_clause is missing the `window_clause` defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

If the order_by_clause is present the `window_clause` defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. For RANK and DENSE_RANK the order_by_clause is mandatory.

OVER() is equivalent to OVER(PARTITION BY NULL RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING). This is also equivalent to a standard SetFunction without GROUP BY.

5.5.3 StringFunction

StringFunctions accept character expressions (strings) as input and compute integers or strings. NULL is returned when one of the operands is NULL.

Transbase V5.3: For UTF-8 databases, all StringFunctions refer to characters, not bytes.

Syntax:

```
StringFunction ::=
      PositionFunction
    | InstrFunction
    | CharacterLengthFunction
    | UpperFunction
    | LowerFunction
    | TrimFunction
    | SubstringFunction
    | ReplaceFunction
    | ReplicateFunction
    | TocharFunction
```

Note: Beside the above summarized string functions, there existst concatenation of strings, denoted with infix operator `'+'` or `'||'` (see Chapter *Expression*).

5.5.3.1 PositionFunction

The POSITION function searches a string inside another string and computes the position of its first occurrence if any.

Syntax:

```
PositionFunction ::=
    POSITION ( Search IN Source )

Search ::= Expression

Source ::= Expression
```

Explanation: Search and Source must be CHAR or BINCHAR. Resulttype is INTEGER.

If *Search* is the empty string, the function returns 1.

In general, the function checks whether *Search* occurs as substring in *Source*: if not it returns 0 else the position of the first occurrence (positions start with 1).

The search is made case sensitive. No wildcard mchanism is supported.

Transbase V5.3: For UTF-8 databases, the position returned is given in characters, not bytes.

Example:

```
POSITION ( 'ssi' IN 'Mississippi' )     -->  3
POSITION ( 'mis' IN 'Mississippi' )     -->  0
```

5.5.3.2 InstrFunction

The INSTR function searches a string inside another string. It provides a superset of the functionality of POSITION.

Syntax:

```
InstrFunction ::=
    INSTR ( Source, Search, [ Startpos [, Occur ] ])

Search ::= Expression

Source ::= Expression

Startpos ::= Expression

Occur ::= Expression
```

Explanation: Search and Source must be CHAR or BINCHAR. Startpos and Occur must be arithmetic expressions of type INTEGER. Resulttype is INTEGER.

Default values for Startpos and Occur are 1.

Let s be the value of Startpos and o be the value of Occur.

In general, the function searches the string "Search" in "Source" starting the search on the s-th character of "Source" (for s >= 1). If o > 1 then the o-th occurrence of "Search" is searched.

If s <= -1 then the search is made backwards starting with the |s|-th character counted relative to the end of "Source".

The search is made case sensitive. No wildcard mchanism is supported. The function returns 0 if the search is unsuccessful else the position of the detectedf substring.

Transbase V5.3: For UTF-8 databases, the position returned is given in characters, not bytes.

Example:

```
INSTR ('Mississippi','ssi )         -->  3
INSTR ('Mississippi','ssi, 4 )      -->  6
INSTR ('Mississippi','ssi, -1 )         -->  6
INSTR ('Mississippi','ssi, -1, 2 )      -->  3
```

5.5.3.3 CharacterLengthFunction

The CHARACTER_LENGTH function computes the length of a string value in characters.

Syntax:

```
CharacterLengthFunction ::=
    CHARACTER_LENGTH( Source )
Source ::= Expression
```

Explanation: The following expressions are equivalent to ensure compatibility with SQL2:

```
CHARACTER_LENGTH ( Source )
SIZE OF ( Source )
```

5.5.3.4 UpperFunction, LowerFunction

The UPPER and LOWER function maps uppercase letters to lowercase letters and vice versa.

Syntax:

```
UpperFunction ::=
    UPPER ( Source )

LowerFunction ::=
    LOWER ( Source )

Source ::= Expression
```

Explanation: Source must be CHAR or BINCHAR. Resulttype is same as Sourcetype.

The function UPPER (LOWER) replaces all lowercase (uppercase) letters by corresponding uppercase (lowercase) letters and leaves all other characters unchanged.

Transbase V5.3: Which characters are mapped to their lowercase (uppercase) equivalent, is determined by the Locale setting of the database. All ASCII characters (a..z and A..Z) are always mapped. When e.g. the Locale setting of the database is a German one, German Umlaut characters are mapped.

Example:

```
UPPER ( 'Line:24' )        -->  'LINE:24'
```

5.5.3.5 TrimFunction

The TRIM function eliminates in a string leading and/or trailing characters belonging to a specifiable character set.

Syntax:

```
TrimFunction ::= TrimFunc | LtrimFunc | Rtrimfunc

TrimFunc ::=
    TRIM ( [ [ Trimspec] [ Trimset ] FROM ]  Source )

Trimspec ::=
    LEADING | TRAILING | BOTH

Trimset, Source  ::= Expression

LtrimFunc ::=
    LTRIM ( Source [ , Trimset ] )

RtrimFunc ::=
    RTRIM ( Source [ , Trimset ] )
```

Explanation: Trimset, Source must be CHAR or BINCHAR. Resulttype is same as Sourcetype.

FROM must be specified if and only if at least one of *Trimset* or *Trimspec* is specified.

If *Trimspec* is not specified, BOTH is implicit.

If *Trimset* is not specified, a string consisting of one ' ' (blank) is implicit.

Depending on whether LEADING, TRAILING, BOTH is specified, the TRIM function delivers a string which is made from *Source* by eliminating all leading characters (trailing characters, leading and trailing characters, resp.) which are in *Trimspec*.

Error occurs if *Trimset* is the empty string.

LTRIM(Source,Trimset) equals TRIM(LEADING Trimset FROM Source).

RTRIM(Source,Trimset) equals TRIM(TRAILING Trimset FROM Source).

Example:

```
TRIM ( '   Smith   ' )                  -->  'Smith'
TRIM ( ' ' FROM '   Smith   ' )         -->  'Smith'
TRIM ( BOTH ' ' FROM '   Smith   ' )    -->  'Smith'
TRIM ( LEADING ' ' FROM '   Smith   ' ) -->  'Smith   '
TRIM ( 'ijon' FROM 'joinexpression' )   -->  'express'
```

5.5.3.6 SubstringFunction

The SUBSTRING function extracts a substring from a string value.

Syntax:

```
SubstringFunction ::= SubstringFunc | SubstrFunc
SubstringFunc ::=
    SUBSTRING ( Source FROM Startpos  [ FOR  Length ] )
SubstrFunc ::=
    SUBSTR ( Source , Startpos  [, Length ] )

Source, Startpos, Length ::=
    Expression
```

Explanation: *Source* must be CHAR or BINCHAR. *Startpos* and *Length* must be arithmetic. Resulttype is same as Sourcetype.

The function constructs a string which results from *Source* by extracting *Length* letters beginning with the one on position *Startpos*. If *Length* is not specified or is larger than the length of the substring starting at *Startpos*, the complete substring starting at *Startpos* constitutes the result.

If *Startpos* is less equal zero then *Length* (if specified) is set to Length + Startpos and Startpos is set to 1 .

Error occurs if *Length* is specified and less than zero.

If *Startpos* is larger than the length of *Source*, the result is the empty string.

SUBSTR(Source,Startpos,Length) is equivalent to SUBSTRING(Source FROM Startpos FOR Length)

Transbase V5.3: For UTF-8 databases, Source, Startpos and Length are specified in characters, not bytes.

Example:

```
SUBSTRING ( 'joinexpression' FROM 5 )        --> 'expression'

SUBSTRING ( 'joinexpression' FROM 5 FOR 7)   --> 'express'

SUBSTRING ( 'joinexpression' FROM 5 FOR 50)  --> 'expression'

SUBSTRING ( 'joinexpression' FROM -2 FOR 6)  --> 'join'
```

5.5.3.7 ReplaceFunction

The REPLACE function replaces substrings or characters in a string.

Syntax:

```
ReplaceFunction ::=
    REPLACE ( Subs1  BY  Subs2  IN  Source [ , Subsspec ] )

Subsspec ::=  WORDS | CHARS

Source, Subs1, Subs2 ::=
    Expression
```

Explanation: *Source*, *Subs1*, *Subs2* must be CHAR or BINCHAR. Resulttype is same as Sourcetype.

The function constructs from *Source* a result string by substituting certain substrings in *Source*.

If *Subsspec* is not defined or defined as WORDS, then all occurrences of *Subs1* are replaced by *Subs2* (after substitution, the inserted string *Subs2* is not further checked for substitution).

If *Subsspec* is defined as CHARS, then *Subs1* and *Subs2* must have same length and each character in *Source* which is equal to the i-th character in *Subs1* for some i is replaced by the i-th character of *Subs2*.

Subs1 must have length greater equal to 1.

Example:

```
REPLACE ( 'iss' BY '' IN 'Mississippi' )          --> 'Mippi'

REPLACE ( 'act' BY 'it' IN 'transaction' )        --> 'transition'

REPLACE ( 'TA' BY 'ta' IN 'TransAction' , CHARS ) --> 'transaction'
```

5.5.3.8 ReplicateFunction

The REPLICATE function replicates a string a specified number of times.

Syntax:

```
ReplicateFunction ::=
    REPLICATE ( Source , Times )

Source , Times  ::=
    Expression
```

Explanation: *Source* must be CHAR or BINCHAR. *Times* must be arithmetic. Resulttype is same as Sourcetype.

The function constructs a result string by concatenating *Source* t times where t is the value of *Times*. Error occurs if t is less than zero.

Example:

```
    REPLICATE ( 'a' , 3 )    -->  'aaa'
```

5.5.3.9 TocharFunction

The Tocharfunction is a shorthand notation for a CAST operator to CHAR(*) .

Syntax:

```
TocharFunction ::=
    TO_CHAR ( expr )
```

Explanation: TO_CHAR(expr) is equivalent to CAST (expr AS ChAR(*)).

5.5.4 SignFunction

Computes the sign of a numerical expression.

Syntax:

```
SignFunction ::= SIGN(Expression)
```

Explanation: Expression must be of numerical type. The function returns -1, 0, 1 depending on whether the value of the expression is negative, 0 or positive.

Example:

```
SIGN(13)  yields 1.
SIGN(0)  yields 0.
SIGN(-13)  yields -1.
```

5.5.5 ResultcountFunction

Numbers result tuples of a select query.

Syntax:

```
ResultcountFunction ::= RESULTCOUNT
```

Explanation: RESULTCOUNT is an unary function which can be used in the SELECT list of the outermost SELECT block. Only one SELECT block of outermost level must exist, i.e. no UNION, INTERSECT etc. are allowed to combine several SELECT blocks of the outermost level. RESULTCOUNT forms one column in the result table and has successive values of 1,2,3 etc.

Example:

```
SELECT RESULTCOUNT, suppno, partno
FROM quotations
```

5.5.6 SequenceExpression

Performs a nextval or currval operation on a sequence.

Syntax:

```
SequenceExpression ::= Sequencename.NEXTVAL | Sequencename.CURRVAL
```

Explanation: See explanations in the section 3.14.

Example:

```
INSERT INTO T VALUES(S.NEXTVAL,13,100);
```

5.5.7 ConditionalExpression

ConditionalExpressions compute one of several values depending on one or several conditions. They are introduced by keywords IF, CASE, DECODE, COALESCE, NVL, NULLIF.

Syntax:

```
ConditionalExpression ::=
      IfExpression
    | CaseExpression
    | DecodeExpression
    | CoalesceExpression
    | NvlExpression
    | NullifExpression
```

5.5.7.1 IfExpression

The *IfExpression* is the simplest *ConditionalExpression*. It computes one of 2 values depending on one condition.

Syntax:

```
IfExpression ::=
    IF SearchCondition
    THEN Expression
    ELSE Expression
    FI
```

Explanation: The result value of the *IfExpression* is determined by the *SearchCondition*: if the *SearchCondition* evaluates to TRUE then the value of the *Expression* in the THEN-part is delivered else the value of the *Expression* in the ELSE-part.

The data types of the two *Expression*s must be compatible. If the types differ then the result is adapted to the higher level type.

Example:

```
SELECT suppno, partno,
    price * IF suppno = 54 THEN 1.1 ELSE 1 FI
FROM quotations

SELECT suppno, partno,
    price * IF suppno = 54
        THEN 1.1
        ELSE
        IF suppno = 57 THEN 1.2 ELSE 1 FI
    FI
FROM quotations
```

Note that the second example is easier formulated by a CASE expression.

5.5.7.2 CaseExpression

The *CaseExpression* is the most general *ConditionalExpression*. It comes in the variants simple CASE and searched CASE.

Syntax:

```
CaseExpression ::=
    SearchedCaseExpression
  | SimpleCaseExpression
```

```
SearchedCaseExpression ::=
    CASE
    SearchedWhenClause [ SearchedWhenClause ] ...
    ELSE  Result
    END

SeachedWhenClause ::=
    WHEN  SearchCondition  THEN Result

SimpleCaseExpression ::=
    CASE  CaseOperand
    SimpleWhenClause [ SimpleWhenClause ] ...
    ELSE  Result
    END

SimpleWhenClause ::=
    WHEN  WhenOperand  THEN Result

CaseOperand, WhenOperand, Result ::=
    Expression
```

Explanation: The *SearchedCaseExpression* successively evaluates the *SearchConditions* of its *SearchedWhenClauses* and delivers the *Result* of the THEN clause of the first *SearchedWhenClause* whose condition evaluates to TRUE. If no condition evaluates to TRUE then the Result of the ELSE clause is delivered if it exists else NULL.

The *SimpleCaseExpression* is equivalent to a *SearchedCaseExpression* with multiple *SearchConditions* of the form

CaseOperand = WhenOperand

where the *WhenOperand* of the i-th *SearchCondition* is taken from the i-th *SimpleWhenClause* and the THEN clauses and ELSE clause (if existent) are identical.

For both variants, all *Result* expressions in the THEN clauses as well as the *Result* of the ELSE clause (if existent) must be type compatible. The result type of the *CaseExpression* is the highest level type of all participating result expressions.

For the *SimpleCaseExpression*, the types of the *CaseOperand* and all *WhenOperands* must be type compatible.

Example:

```
UPDATE quotations
SET price = price * CASE
    WHEN price > 25      THEN  1.5
    WHEN price > 19.5    THEN  1.4
    WHEN price > 5       THEN  1.3
    WHEN price > 1       THEN  1.2
    ELSE   1.1
    END

SELECT suppno, partno, price * CASE
    WHEN suppno = 54 THEN 1.1
    WHEN suppno = 57 THEN 1.2
    ELSE    1
    END
FROM quotations

SELECT suppno, partno, price * CASE  suppno
    WHEN 54 THEN 1.1
    WHEN 57 THEN 1.2
    ELSE    1
    END
FROM quotations
```

5.5.7.3 DecodeExpression

The *DecodeExpression* is an alternative way to denote a CaseExpression of variant SimpleCaseExpression.

Syntax:

```
DecodeExpression ::=
    DECODE ( CompareExpr , MapTerm [ , MapTerm ] ...
        [ , DefaultExpr ] )

MapTerm ::=
    WhenExpr , ThenExpr

CompareExpr, WhenExpr, ThenExpr, DefaultExpr,  ::=
    Expression
```

Explanation: The *CompareExpr* is successively compared with the *WhenExpr*s of the *MapTerm*s. If the comparison matches then the corresponding *ThenExpr* is delivered as result. If none of the comparisons matches then *DefaultExpr* is delivered as result if specified otherwise the Null value. All expressions must be type compatible. The result type is the highest level type of all participating expressions.

Example:

```
SELECT suppno, partno, price *
        DECODE (suppno, 54, 1.1, 57, 1.2, 1)
FROM quotations
```

5.5.7.4 CoalesceExpression, NvlExpression, NullifExpression

COALESCE and NVL are shorthand notations for a CASE or IF which maps an *Expression* from the NULL value to a defined value. The NULLIF is a shorthand notation for a CASE or IF which maps an expression from a defined value to the NULL value.

Syntax:

```
CoalesceExpression ::=
    COALESCE ( ExpressionList )
ExpressionList ::=
    Expression [, Expression ] ...
NvlExpression ::=
    NVL ( Expression , Expression )
NullifExpression ::=
    NULLIF ( Expression , Expression )
```

Explanation: All involved expressions must be of compatible types. The result type is the highest level type of the result expressions.

COALESCE delivers the first expression which does not evaluate to NULL if there exists such an expression otherwise NULL.

Thus it is equivalent to an expression of the form

```
CASE
    WHEN x1  IS NOT NULL  THEN  x1
    WHEN x2  IS NOT NULL  THEN  x2
    ...
```

```
    ELSE NULL
    END
```

Note that with COALESCE, each involved expression is denoted only once in contrast to an equivalent CASE or IF construction. Therefore, in general, the COALESCE runs faster.

NVL is equivalent to COALESCE but restricted to 2 arguments.

NULLIF delivers NULL if the comparisons of both Expressions evaluates to TRUE else it delivers the value of the first Expression.

Thus it is equivalent to an expression of the form

IF x1 = x2 THEN NULL ELSE x1 FI

NULLIF in general runs faster than an equivalent CASE or IF construction because the first expression is evaluated only once. It is most often used to map an explicitly maintained non-NULL default value of a field back to its NULL semantics when used for computation.

5.5.8 TimeExpression

A TimeExpression is an expression which is based on value of type DATETIME or TIMESPAN

Syntax:

```
TimeExpression ::=
      [ Selector OF ] { TimePrimary | Constructor }

Selector ::=
      WEEKDAY | YY | MO | DD | HH | MI | SS | MS

TimePrimary ::=
      DatetimeLiteral
    | TimespanLiteral
    | FieldReference
    | HostVarInd
    | CURRENTDATE
    | SYSDATE
    | (Expression)
    | (SubTableExpression)
    | SetFunction
```

```
    | TruncFunction
    | ConditionalExpression

Constructor ::=
      CONSTRUCT Timetype ConstructList

Timetype ::=
      { DATETIME | TIMESPAN } [ RangeQualifier ]

RangeQualifier ::=
      < see Chapter DataType >

ConstructList ::=
      ( Constituent [ , Constituent] ...)

Constituent ::=
      Expression | SubTableExpression

TruncFunction ::=
      TRUNC(Expression)
```

Explanation: For all semantics see Chapter 9 (The Data Types Datetime and Timespan).

Note that a selector as well as a constructor binds more strongly than a CAST operator (see also Precedence of Operators).

Example:

```
DATETIME[YY:MS](1989-6-8 12:30:21.032)
CURRENTDATE
HH OF CURRENTDATE
WEEKDAY OF CURRENTDATE
CONSTRUCT TIMESPAN(:year,:month,:day)
```

5.5.9 SizeExpression

The SIZE [OF] Operator computes the size (length) of a CHAR or BLOB Expression.

Syntax:

```
SizeExpression ::=
    SIZE [ OF ] Literal
  | SIZE [ OF ] FieldReference
  | SIZE [ OF ] HostVarInd
  | SIZE [ OF ] User
  | SIZE [ OF ] (Expression)
  | SIZE [ OF ] (SubTableExpression)
  | SIZE [ OF ] SetFunction
  | SIZE [ OF ] ConditionalExpression
  | SIZE [ OF ] BlobExpression
```

Explanation: The resulting type of the argument of the SIZE operator must be CHAR(*), (VAR)CHAR(p), BINCHAR(*), BINCHAR(p), BITS(*), BITS(p) or BLOB. The resulting type of the operator is INTEGER.

If the argument of the operator is the NULL value, then the operator delivers NULL. Otherwise the operator delivers a value that denotes the size (in bytes, for BITS in bits) of its argument. If the argument is CHAR(*) or (VAR)CHAR(p) then the trailing `\0` is not counted. If the argument is BINCHAR(*) or BINCHAR(p) then the length field is not counted. If the argument is BLOB then the number of bytes that the BLOB object occupies is delivered.

Note also the strong binding of the SIZE operator (see Precedence of Operators).

Example:

```
SIZE OF 'abc'                 --> 3
SIZE OF 0x0a0b0c              --> 3
SIZE OF bl                    --> length of the BLOB column bl
SIZE OF bl SUBRANGE (1,10)    --> 10 if bl is at least 10 long.
```

5.5.10 BlobExpression

A BlobExpression delivers a BLOB value or a subrange of a BLOB value.

Syntax:

```
BlobExpression ::=
    FieldReference [ SUBRANGE ( Lwb , Upb ) ]

Lwb ::= Expression

Upb ::= Expression
```

Explanation: The field must be of type BLOB, *Lwb* and *Upb* must be of type TINYINT, SMALLINT or INTEGER. The resulting value of *Lwb* and *Upb* must not be less or equal 0.

If SUBRANGE is not specified, then the resulting value is the BLOB object of the denoted *Field*. If one of *Field*, *Lwb* or *Upb* is the NULL value then the resulting value is also the NULL value. Otherwise the BLOB object restricted to the indicated range is delivered.

The smallest valid *Lwb* is 1. If *Upb* is greater than (SIZE OF Field) then it is equivalent to (SIZE OF Field).

If the value of *Upb* is less than the value of *Lwb* then a BLOB object of length 0 is delivered.

Example: Let bl a BLOB object of length 100:

```
bl SUBRANGE (1,1)       --> first byte of bl as BLOB

bl SUBRANGE (1,SIZE bl) --> bl as it is

bl SUBRANGE (50,40)     --> empty BLOB object
```

5.5.11 ODBC_FunctionCall

Syntax:

```
ODBC_FunctionCall ::=
    { fn FuncId(..) }

FuncId ::= Identifier
```

Explanation: By the ODBC function call syntax, an embedding of the ODBC functions is provided to the Transbase SQL syntax.

5.5.12 UserDefinedFunctionCall

Syntax:

```
UserDefinedFunctionCall ::=
    FuncName ( ExpressionList )

FuncName ::= Identifier
```

Explanation: A UserDefinedFunction (written as function returning one value) can be called at any place in the SQL statement where one value is accepted as result. Parameters of the function may be Expressions delivering one value (including dynamic parameters '?' supplied by the application at runtime).

Example:

```
SELECT sqrt(field)
FROM T
WHERE field > 0
```

5.6 SearchCondition

*SearchCondition*s form the WHERE-clause and the HAVING-clause of a *Select-Expression* and return a boolean value for each tuple and group, resp. They also appear in *ConditionalExpressions* to choose one of two *Expression*s.

Syntax:

```
SearchCondition ::=
    [NOT] Predicate [ Boolop [NOT] Predicate ] ... ]
Boolop ::=
    AND | OR
```

Explanation: The precedence of operators is: 'NOT' before 'AND' before 'OR' (see Precedence of Operators). Additional parentheses may be used as usually to override precedence rules (see Predicate).

Example:

```
(suppno = 54 OR suppno = 57) AND qonorder > 0

NOT ((suppno <> 54 AND suppno <> 57) OR qonorder <= 0)
```

If no null values are involved, these two SearchConditions are equivalent.

5.7 HierarchicalSearchCondition

In addition to the *SearchCondition* in the WHERE clause hierarchical data can be queried using *HierarchicalSearchCondition*s.

Syntax:

```
HierarchicalSearchCondition ::=
    [START WITH SearchCondition]
    CONNECT BY [NOCYCLE] ConnectByCondition

ConnectByCondition ::=
    <Predicate using HierarchicalExpression>

HierarchicalExpression ::=
    | LEVEL
    | CONNECT_BY_ISCYCLE
    | CONNECT_BY_ISLEAF
    | PRIOR Expression
    | CONNECT_BY_ROOT Expression
    | SYS_CONNECT_BY_PATH(Expression, StringLiteral)
```

Explanation: START WITH defines the set of root rows for a hierarchical query. It is formulated as an SearchCondition without *HierarchicalExpression*s. If this optional clause is omitted every row the set defined by the FROM clause is considered as root row.

CONNECT BY defines the relationship between the rows of a hierarchy. References to a prior or root rows or other hierarchical relations can be phrased using *HierarchicalExpression*s. NOCYCLE controls the behaviour if a cycle in the hierarchical data is encountered. Usually, if a cycle in hierarchical data is found, then this will result in an error, since otherwise the query would produce an infinite loop of tuples. If NOCYCLE is specified cycles are ignored, i.e. the algorithm will not follow a path that leads to a tuple that has already been printed.

*HierarchicalExpression*s can be used like FieldReferences throughout the current and outer query blocks, except in the START WITH clause.

LEVEL stands for the hierarchy level of the current row, i.e. a root node is on LEVEL 1, its successors are on LEVEL 2 and so on.

CONNECT_BY_ISCYCLE and CONNECT_BY_ISLEAF return integer values. Here a value of 1 indicates that the current row is the beginning of a cycle in a hierarchy or a leaf in the hierarchy, resp.

PRIOR and CONNECT_BY_ROOT are unary operators that indicate that *Expression* refers to FieldReferences from the prior or root row.

SYS_CONNECT_BY_PATH is a built-in function that calculates the path from the root row to the current row by concatenating the results of *Expression* for every visited predecessor, separating each with *StringLiteral*.

Example: Please consider the following hierarchical data sample and queries.

id	parent_id
1	2
2	1
3	2
4	2
5	6
6	5
7	7

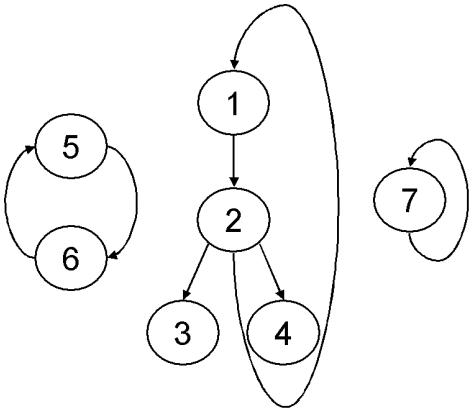

Figure 5.1: Hierarchical data and graph

```
SELECT id, parent_id, LEVEL "level",
CONNECT_BY_ISLEAF isleaf, CONNECT_BY_ISCYCLE iscyle,
SYS_CONNECT_BY_PATH(id,'/') path
FROM hierarchy
WHERE level < 4
START WITH id = 1
CONNECT BY NOCYCLE parent_id = PRIOR id
```

id	parent_id	level	isleaf	iscylce	path
1	2	1	0	0	/1
2	1	2	0	1	/1/2
3	2	3	1	0	/1/2/3
4	2	3	1	0	/1/2/4

At each point in time of the depth-first search, there exists a "current row". At the beginning, one of the root rows satisfying the START WITH condition is chosen as "current row". In this example it is the row with id 1. To find the next row, the ConnectByCondition is evaluated whereby the PRIOR expressions are those which refer to the current row (thus representing defined and constant value for the actual search) and the remaining expressions are treated like in a standard search. In the example, given the row id 1 as the current row, the search condition "parent_id = PRIOR id" effectively is "parent_id = 1" as PRIOR id evaluates to 1 in the current row. The first result tuple of this search then becomes the new "current row" and the algorithm proceeds depth-first down the hierarchy until no more successors are found. If a search on one level delivers more than one result tuple, the remaining tuples successively become the "current row" after the recursive searches have finished.

The result indicates that a cycle begins in row 2 since it leads back to the root row. This is also why the NOCYCLE option is required. Rows 3 and 4 are leaf rows.

```
SELECT id, parent_id, level "level",
SYS_CONNECT_BY_PATH(id,'/') path
FROM hierarchy
START WITH id > 4
CONNECT BY NOCYCLE parent_id = PRIOR id
```

id	parent_id	level	path
5	6	1	/5
6	5	2	/5/6
6	5	1	/6
5	6	2	/6/5
7	7	1	/7

Here query rows 5, 6, and 7 satisfy the START WITH condition. The query result is equivalent to a union of three queries with each of these rows successively acting as root rows.

5.8 Predicate

*Predicate*s are the building units for *SearchCondition*s.

Syntax:

```
Predicate ::=
      (SearchCondition)
    | ComparisonPredicate
    | BetweenPredicate
    | LikePredicate
    | MatchesPredicate
    | ExistsPredicate
    | QuantifiedPredicate
    | NullPredicate
    | FulltextPredicate
```

5.8.1 ComparisonPredicate

A ComparisonPredicate compares two values or two sets of tuples or checks one value/tuple to be in a set of values/tuples.

Syntax:

```
ComparisonPredicate ::=
      ValueCompPredicate
    | SetCompPredicate
    | InPredicate
```

5.8.2 ValueCompPredicate

A ValueCompPredicate compares two values.

Syntax:

```
ValueCompPredicate ::=
    Expression  ValCompOp  Expression

ValCompOp ::=
    < | <= | = | <> | > | >=
```

Explanation: The meaning of the operators are:

`<`	less than
`<=`	less than or equal to
`=`	equal
`<>`	not equal
`>`	greater than
`>=`	greater than or equal to

The data types of the *Expression*s must be compatible. If *TableExpression*s are used, they must deliver a single value.

The comparison operators are defined for all data types.

If two character sequences (strings) with different length are compared, then the shorter string is padded with the space character ' ' up to the length of the longer string.

For the following examples of correct *ValueCompPredicate*s, assume that q is a correlation name for the table quotations.

Example:

```
suppno < 54

price * qonorder < 100.50

q.price >
(SELECT AVG (price)
FROM quotations
WHERE partno = q.partno)

(SELECT MAX(price) - MIN(price)
FROM quotations
WHERE partno = q.partno)
>
(SELECT AVG (price)
FROM quotations
WHERE partno = q.partno) * 0.5
```

The last example would be a suitable *SearchCondition* to find out partnos from tuples q in quotation with a big variance in price offerings.

5.8.3 SetCompPredicate

A *SetCompPredicate* compares two sets of tuples.

Syntax:

```
SetCompPredicate ::=
    (SubTableExpression) SetCompOp (SubTableExpression)

SetCompOp ::=
    [NOT]  SUBSET  [OF]
    | =
    | <>
```

Explanation: Let q1 and q2 be the two *SubTableExpression*s and s1 and s2 their result sets of tuples, resp. q1 and q2 must be compatible, i.e. their result sets must have the same arity n ($n \geq 1$) and each pair of types of the corresponding fields must be type compatible (see chapter 2.2 Data Types). Two n-ary tuples t1 and t2 *match* if they have the same values on corresponding fields. `q1 SUBSET q2`

yields TRUE if for each tuple t1 from Result of q1 there is a matching tuple t2 in Result set of q2.

The following notations are equivalent:

```
q1 NOT SUBSET q2
NOT (q1 SUBSET q2)
```

Parentheses can be omitted, see Precedence of Operators `q1 = q2` yields TRUE if s1 and s2 are identical, i.e. for each tuple t1 in s1 there is a matching tuple t2 in s2 and vice versa.

The following notations are equivalent:

```
q1 <> q2
NOT q1=q2
```

Note: Duplicate tuples in any of s1 or s2 do not contribute to the result, i.e. sets are treated in the mathematical sense in all set comparison operators.

Note: The operator CONTAINS (the symmetric counterpart of SUBSET) is obsolete and should only be used for fulltext predicates from now on.

Example: List all supplier numbers who deliver (at least) the same parts and price offerings as supplier 54

```
SELECT DISTINCT suppno
FROM quotations q
WHERE
    (SELECT partno, price
    FROM quotations
    WHERE suppno = 54)
        SUBSET
    (SELECT partno, price
    FROM quotations
    WHERE suppno = q.suppno)
```

5.8.4 InPredicate

The InPredicate checks if an explicitly specified tuple is in a set of tuples or checks if a value is in a set of values.

Syntax:

```
InPredicate ::=
    ValueInPredicate  |  TupleInPredicate

ValueInPredicate ::=
    Expression [NOT] IN
    { (ExpressionList) | (SubTableExpression) }

ExpressionList ::=
    Expression [, Expression ] ...

TupleInPredicate ::=
    Tuple [NOT] IN (SubTableExpression)

Tuple ::=
    LeftBr  ExpressionList  RightBr

LeftBr ::= [

RightBr ::= ]
```

Explanation: A *ValueInPredicate* checks a value against a set of values. If an *ExpressionList* is specified, the predicate yields TRUE if the value of the left hand *Expression* is equal to one of the values of the *ExpressionList*. If a *SubTableExpression* is specified it must deliver unary tuples which then are interpreted as a set of values like above.

A *TupleInPredicate* checks a tuple against a set of tuples. It yields TRUE if the left hand tuple matches one of the result tuples of the *SubTableExpression*. Compatibility rules for *Tuple* and *TableExpression* are analogous to those of *SetCompPredicate*.

Note: The number of *Expression*s contained in an *ExpressionList* is limited to 40. The notation x NOT IN y is equivalent to NOT x IN y

Example:

```
SELECT *
FROM suppliers
WHERE suppno IN (54,61,64)
```

Example:

```
SELECT * FROM suppliers
WHERE suppno IN
    (SELECT suppno
    FROM quotations)
```

Example: List suppliers who deliver at least one part for the same price as supplier 57

```
SELECT DISTINCT suppno FROM quotations
WHERE  [partno, price] IN
    (SELECT partno, price
    FROM quotations
    WHERE suppno = 57)
```

5.8.5 BetweenPredicate

The *BetweenPredicate* tests a value against an interval of two values. Each of the two interval boundaries can be specified as inclusive or exclusive.

Syntax:

```
BetweenPredicate ::=
        Expression [NOT] BETWEEN
        Expression [BetweenQualifier]
        AND
        Expression [BetweenQualifier]

BetweenQualifier ::=
    INCLUSIVE | EXCLUSIVE
```

Explanation: If a *BetweenQualifier* is omitted it is equivalent to INCLUSIVE. The notation `e1 BETWEEN e2 AND e3` therefore is equivalent to `e1>=e2 AND e1 <=e3`. The notation `e1 BETWEEN e2 EXCLUSIVE AND e3` is equivalent to `e1>e2 AND e1<=e3`. The notation `e1 NOT BETWEEN e2 AND e3` is equivalent to `NOT (e1 BETWEEN e2 AND e3)`

Example:

```
price BETWEEN 0.10 AND 0.30

q.price NOT BETWEEN
    (SELECT MAX (price) FROM quotations
    WHERE partno = q.partno) * 0.8  EXCLUSIVE
AND
    (SELECT MIN (price) FROM quotations
    WHERE partno = q.partno) * 1.2  EXCLUSIVE
```

5.8.6 LikePredicate

The *LikePredicate* tests a string value against a pattern.

Syntax:

```
LikePredicate ::=
    Expression
    [NOT] LIKE Sensspec
    Pattern [ ESCAPE EscapeChar ]

Sensspec ::=
    SENSITIVE | INSENSITIVE

Pattern ::=
    Expression

EscapeChar ::=
    Expression
```

Explanation: All specified *Expression*s must be of character type. The type of *EscapeChar* must be CHAR (1), i.e. a character string of byte length 1.

Transbase V5.3: For UTF-8 databases, *EscapeChar* is restricted to be a single character with Unicode value less than 128 (whose byte length is 1).

Note that all Expressions including the *Pattern* need not be constants but may also be calculated at runtime.

The result of *Pattern* is interpreted as a search pattern for strings where two special characters have the meaning of wild cards:

- The percent sign % matches any string of zero or more characters
- The underscore sign _ matches any single character

Transbase V5.3: Please note that for UTF-8 databases pattern matching is done character-wise (not byte-wise). In particular, a single character can consist of more than one byte.

If *EscapeChar* is specified (let its value be c) then all occurrences of wild card characters in *Pattern* which are preceeded by a c are not interpreted as wild card characters but as characters in their original meaning and the *EscapeChar* c is not interpreted as part of the pattern in these occurences.

If *Sensspec* is not specified or is specified with SENSITIVE then the search pattern is interpreted case sensitive, otherwise the search is performed case insensitive.

Transbase V5.3: The insensitive character comparison depends on the Locale setting of the database.

The notations are equivalent:

```
s NOT LIKE p ESCAPE c
NOT (s LIKE p ESCAPE c)
```

Example:

```
description LIKE 'B%'

description LIKE INSENSITIVE '%_r'

description LIKE '%#%' ESCAPE '#'
```

The first example yields TRUE for values in description which begin with 'B', the second analogously for all values which end with 'r' or 'R' and have at least 2 characters. The third example yields TRUE for all values which end with the percent sign.

Note: If no wildcard is used in the pattern, e.g. `description LIKE 'xyz'` then this expression is *not* equivalent to `description = 'xyz'` because the string comparison ignores trailing blanks whereas the LIKE operator is sensitive with respect to trailing blanks.

5.8.7 MatchesPredicate, Regular Pattern Matcher

The *MatchesPredicate* tests a string value against a pattern denoted as a regular expression.

Syntax:

```
MatchesPredicate ::=
    Expression [NOT] MATCHES Sensspec
    RegPattern [ ESCAPE EscapeChar ]

Sensspec ::=
    SENSITIVE | INSENSITIVE

RegPattern ::=
    Expression

EscapeChar ::=
    Expression
```

Explanation: All specified Expressions must be of character type. The type of *EscapeChar* must be CHAR (1), i.e. a string of byte length 1.

Transbase V5.3: For UTF-8 databases, *EscapeChar* is restricted to be a single character with Unicode value less than 128 (whose byte length is 1).

The result of *RegPattern* is interpreted as a regular expression. Regular expressions are composed of characters and metacharacters. Metacharacters serve as operands for constructing regular expressions. The following characters are metacharacters: `( ) { } [ ] * . , ? + - |`

In all following examples, the patterns and values are written in *CharLiteral* notation (i.e. with surrounding single quotes).

Characters and Character Classes: Patterns may be composed of characters and character classes. A character in a pattern matches itself. For example, the pattern 'xyz' is matched by the value 'xyz' and nothing else (in case sensitive mode). A character class is either a dot sign (`'.'`) or a construct in square brackets `[]`. The dot sign is matched by any character. For example, the pattern 'x.z' is matched by values 'xaz', 'xbz', etc. A character class in `[]` is matched by any character listed in `[]`. The list is either a sequence of single characters like in `[agx]`, or it is a character range like `[a-z]` as a shorthand notation for all characters between a and z (in machine code), or it is a combination of both. For

example, `[ad-gmn]` is matched by any of the characters a, d, e, f, g, m, n. Note that blanks would be interpreted as matchable characters, so don't write `[a b]` or `[ ab ]` if you mean `[ab]`. It is an error to specify character ranges like `[c-a]` where the machine code of the upperbound character is less than that of the first character.

Alternatives: The `|` sign separates alternatives in a pattern. For example, the pattern `abc|yz` is matched by `abc` as well as by `yz`. The implicit character concatenation binds stronger than the alternative sign, so to match either `abcz` or `abyz` one has to specify the pattern `ab(c|y)z` (of course also `abcz|abyz` would work). Note also that character classes are nothing else but a shorthand notation for otherwise possibly lengthy alternatives, so `ab[cy]z` is equivalent to `ab(c|y)z`, too.

Repetition factors: When an asterisk `*` occurs in a pattern, then zero or arbitray many occurrences of the preceding pattern element must occur in the value to match. For example, the pattern `abc*` is matched by `ab`, `abc`, `abcc` etc. All repetition factor operands bind most strongly, so the pattern `(abc)*` must be specified to match `abc`, `abcabc`, etc. The '+' sign means one or more occurrences of the preceding pattern element, so `x+` is identical to `xx*`. The '?' sign means zero or one occurrences. At least n but maximal m occurrences of a pattern element x can be specified by the notation `x{n,m}` where n and m must be integer constants. For example `ag{1,3}z` is matched by `agz`, `aggz`, `agggz`.

Precedence of operands: Three levels of precedence are given, namely the repetition factors which bind stronger than concatenation which binds stronger than the alternative. To overrule the precedence of operators, round precedence brackets can be used as shown in the above examples.

Escaping the metacharacters: Whenever a metacharacter is to stand for itself (i.e. is not wanted in its meta meaning) it must be escaped. If *EscapeChar* is specified (let its value be c) then all occurrences of metacharacters in the pattern which are preceeded by the specified character are not interpreted as metacharacters but as characters in their original meaning and the escape character is not interpreted as part of the pattern in these occurences. For example, in the expression `value MATCHES '\|\?' ESCAPE '\'` the value `|?` matches and any other value does not.

If the escape character is needed as normal character, it must be written twice (normally one can avoid this situation by choosing another escape character).

If *Sensspec* is not specified or is specified with SENSITIVE then the search pattern is interpreted case sensitive, otherwise the search is performed case insensitive.

For example, the expression `s MATCHES INSENSITIVE 'ab.*z'` is equivalent to `s MATCHES SENSITIVE '(a|A)(b|B).*(z|Z)'`

Note that in case of INSENSITIVE, the ranges in character classes are somewhat restricted, i.e. if one of the characters is a lowerbound (upperbound) character then the other must also be a lowerbound (upperbound) character. For example, the ranges `[b-G]` or `[B-g]` are erroneous.

The notations are equivalent:

```
s NOT MATCHES p ESCAPE c
NOT (s MATCHES p ESCAPE c)
```

Note: The *MatchesPredicate* is more powerful than the *LikePredicate* which however is supported for compatibility. A pattern in a *LikePredicate* can be transformed to a regular patterns by substituting each non-escaped `%` by `.*` and each non-escaped `_` by `.`.

5.8.8 ExistsPredicate

The *ExistsPredicate* tests the result of a *SubTableExpression* on emptyness.

Syntax:

```
ExistsPredicate ::=
    EXISTS ( SubTableExpression )
```

Explanation: The predicate evaluates to TRUE if the result of the *SubTableExpression* is not empty.

Example: Which suppliers supply at least 1 part?

```
SELECT suppno, name FROM suppliers s
WHERE EXISTS
    (SELECT *
    FROM quotations
    WHERE suppno = s.suppno)
```

5.8.9 QuantifiedPredicate

A *QuantifiedPredicate* compares one value against a set of values.

Syntax:

```
QuantifiedPredicate ::=
    Expression ValCompOp Quantifier
    ( SubTableExpression )

ValCompOp ::=
    < | <= | = | <> | > | >=

Quantifier ::=
    ALL | ANY | SOME
```

Explanation: The *SubTableExpression* must deliver unary tuples (i.e. a set of values) whose type is compatible with that of *Expression*.

If ALL is specified, the predicate is TRUE if the specified comparison is true for all values delivered by the *SubTableExpression* or if the *SubTableExpression* delivers no value.

If ANY or SOME is specified, the predicate is TRUE if the *TableExpression* delivers at least one value for which the specified comparison is TRUE. Note that ANY and SOME have precisely the same meaning.

Example: List suppliers and parts for which there is no cheaper offering

```
SELECT suppno, partno FROM quotations q
WHERE price <= ALL
    (SELECT price
    FROM quotations
    WHERE partno = q.partno)
```

Example: List all other suppliers

```
SELECT suppno, partno FROM quotations
WHERE price > ANY
    (SELECT price
    FROM quotations
    WHERE partno = q.partno)
```

5.8.10 NullPredicate

A Null-Predicate checks the result of an Expression against the null value.

Syntax:

```
NullPredicate ::=
    Expression IS [NOT] NULL
  | Expression = NULL
  | Expression <> NULL
```

The following notations are equivalent:

```
Expression IS NULL
Expression = NULL
```

The following notations are equivalent, too:

```
Expression IS NOT NULL
NOT (Expression IS NULL)
Expression <> NULL
```

For the semantics of the *NullPredicate* see the chapter Null Values.

5.8.11 FulltextPredicate

On fulltext-indexed fields of type BLOB, VARCHAR(p), CHAR(p) or CHAR(*) , search expressions of type FulltextPredicate can be issued.

Syntax:

```
FulltextPredicate ::=
    FieldName CONTAINS ( FulltextTerm )

FulltextTerm ::=
    FulltextFactor [ OR FulltextFactor ] ...

FulltextFactor ::=
    FulltextPhrase  [ Andnot FulltextPhrase ] ...

Andnot::=
    AND  |  NOT

FulltextPhrase ::=
    ( FulltextTerm )
  | Atom  [ [ DistSpec ]  Atom ] ...
```

```
Atom ::=
    SingleValueAtom
  | MultiValueAtom

SingleValueAtom ::=
    CharLiteral
  | Parameter
  | FtExpression

MultiValueAtom ::=
     ANY ( TableExpression )

DistSpec ::=
    Leftbracket [ MinBetween , ] MaxBetween Rightbracket

Leftbracket ::= [

Rightbracket ::= ]

MinBetween , MaxBetween::=
    <Expression of type Integer>

Parameter ::=  <see Primary>

FtExpression ::=
    <Expression without FieldReference to same block>

CharLiteral ::=
    <literal of type character>
```

Explanation: All explanations are given in the separate chapter on FulltextIndexes.

5.9 Null Values

A tuple may have undefined values (null values) as field-values (if the corresponding *CreateTableStatement* of the table allows it).

A special constant NULL is provided to test a result of Expressions against the null value inside a *SearchCondition*.

Example:

```
price = NULL

price <> NULL
```

The first expression delivers TRUE if the field price is null-valued, it delivers FALSE if the field price has a known value.

If a null-valued field participates in an arithmetic operation (+, -, *, /), the result is again null-valued.

If a null-valued *Expression* participates in a *ValueCompPredicate*, the result of the *ValueCompPredicate* is UNKNOWN.

The evaluation rules for boolean operators are given in tables 5.1, 5.2, 5.3.

NOT	TRUE	FALSE	UNKNOWN
	FALSE	TRUE	UNKNOWN

Table 5.1: NOT operator

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Table 5.2: OR operator

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Table 5.3: AND operator

If the result of a *SearchCondition* is UNKNOWN, it is equivalent to FALSE.

Example: Assume that a field named price is null-valued for a specific tuple, the following *SearchCondition*s effectively evaluate as follows:

```
price < 10.0   -- FALSE
price >= 10.0  -- FALSE
price = null   -- TRUE
```

5.9.1 Further rules for Null-values:

SetFunctions ignore null-values in their input. As far as grouping and duplicate elimination (DISTINCT) is concerned all null-valued fields form one group and are considered equal, resp. In the ORDER BY clause, all NULL values are sorted before any other value.

5.10 SelectExpression (QueryBlock)

A SelectExpression derives tuples from tables in the database. QueryBlock is a synonym for SelectExpression.

Syntax:

```
SelectExpression ::=
    SELECT [ALL | DISTINCT] SelectList
    FROM TableReference  [, TableReference ] ...
    [ WHERE  SearchCondition ]
    [ HierarchicalSearchCondition ]
    [ GROUP BY FieldReference [, FieldReference ] ...
    [ HAVING SearchCondition ]

SelectList ::=
    SelectElem [, SelectElem ] ...

SelectElem ::=
    Expression [ AS FieldName ]
  | [CorrelationName .]*
```

Explanation: Each *TableName* must identify an existing table or view in the database.

Each specified *CorrelationName* must be unique within all *TableName*s and *CorrelationName*s of that FROM clause.

All specified Fields in the GROUP-BY-clause must have a resolution in the given *QueryBlock* (see Rules of Resolution).

Each *TableReference* exports a set of field names (see TableReference). These names can be used in the defining *QueryBlock* to reference fields of the result tuples of the FROM clause.

The *QueryBlock* also exports a set of field names: the i-th field of the block exports name "f" if it is a field reference with fieldname "f" or "q.f" or if it is specified with an "AS f" clause, otherwise the i-th field is "unnnamed".

The result of a QueryBlock is defined by conceptually performing the following steps:

(Step 1) The Cartesian product of the results from the TableReferences in the FROM-clause is constructed.

(Step 2a) All joins defined by the SearchCondition of the WHERE-clause are performed.

(Step 2b) The HierarchicalSearchCondition is processed. A depth-first search is carried out starting with one of the root rows that satisfies the START WITH predicate. For this root row the first child rows satisfying the CONNECT BY condition is selected. Then the hierarchical search will proceed down through the generations of child rows until no more matching rows are found.

(Step 2c) The SearchCondition of the WHERE-clause is applied to all tuples resulting from the previous steps. The result of (2) is the set of tuples which satisfy the SearchCondition and HierarchicalSearchCondition. In addition then the result is sorted in depth-first order with respect to the HierarchicalSearchCondition, if any.

(Step 3) The GROUP-BY-clause partitions the result of (2) into groups of tuples. All tuples with the same values on the specified Fields form one group. Thus, the number of groups is equal to the number of different value combinations on the specified Fields. If a GROUP-BY-clause is specified the following conditions must hold: Asterisk (*) is not permitted in the SELECT-clause. Each Field in the SELECT-clause and in the HAVING-clause which refers to the given QueryBlock (i.e. whose resolution block is the given QueryBlock, see Rules of Resolution) either must be a grouping Field or must be inside a SetFunction whose resolution block is the given QueryBlock.

(Step 4) The SearchCondition of the HAVING-clause is applied to each group. The result of (4) is the set of groups which satisfy the SearchCondition. If no GROUP-BY-clause is specified, the whole set of tuples from the previously executed step forms one group.

(Step 5) Result tuples according to the SELECT-clause are constructed. If neither GROUP-by nor HAVING is specified, each tuple from the previously executed step contributes to one result tuple and each SetFunction which has a local resolution refers to all input tuples of (5). If GROUP-BY and/or HAVING is specified, each group contributes to one result tuple and each local SetFunction is computed separately for each group.

(Step 6) If DISTINCT is specified, duplicate tuples are removed from the result of (5). By default or if ALL is specified, duplicate tuples are not removed.

The asterisk notations in the *SelectList* are shorthand notations for a list of field names. The pure `*` stands for a list of all field names exported by all *TableReference*s of the FROM-clause in their original order. The notation `Identifier.*` stands for a list of all field names of the *TableReference* with *CorrelationName* or *TableName* Identifier. The asterisk notations can be freely mixed among each other and other Expressions.

Updatability: A *SelectExpression* is updatable if all following conditions hold:

- No GROUP BY-clause and no HAVING-clause is specified.
- No DISTINCT is specified in the SELECT-clause.
- The SELECT-clause consists of `*` or each *Expression* in the *ExpressionList* only consists of a *Field* (i.e. no operators, no *SetFunction*) and each *FieldName* only occurs once.
- The WHERE-clause does not contain a subquery.
- The FROM-clause only contains one single *TableReference* and this *TableReference* is updatable (see TableReference).

Note: There exists one more operator called UNGROUP BY. This is a very special operator only used in conjunction with the type BITS(*). To keep the explanation simple here, this is only explained in Chapter The TB SQL Datatypes BITS(p) and BITS(*).

Example: Which parts are delivered by more than 1 supplier?

```
SELECT partno, COUNT (*)
FROM quotations
GROUP BY partno
HAVING COUNT (*) > 1
```

Example: What is the average number of suppliers for a part (2 solutions)?

```
SELECT AVG(cnt) FROM
   (SELECT COUNT (*) AS cnt
   FROM quotations
   GROUP BY partno)
```

```
SELECT AVG(cnt) FROM
(SELECT COUNT (*)
FROM quotations
GROUP BY partno)  (cnt)
```

To introduce a field name "cnt" for the unnnamed field COUNT(*), the first solution directly defines COUNT(*) to have name "cnt", whereas the second solution uses an Alias for the *TableReference* (see *TableReference*)

Example: Which suppliers deliver part 221 (all suppliers and part information delivered):

```
SELECT * FROM quotations q, suppliers s
WHERE q.suppno = s.suppno AND q.partno = 221
```

Example: Which suppliers deliver part 221 (Only suppliers information delivered)

```
SELECT DISTINCT s.* FROM quotations q, suppliers s
WHERE q.suppno = s.suppno AND q.partno = 221
```

5.11 TableExpression, SubTableExpression

A *TableExpression* and *SubTableExpression* construct UNIONs, INTERSECTions and set DIFFerences from the result sets of TableReferences.

SubTableExpression is a slightly restricted form of *TableExpression* (is made up of *SubTableReference*s instead of *TableReference*s).

Syntax:

```
TableExpression ::=
    TableTerm [ UnidiffSpec  TableTerm ] ...
SubTableExpression ::=
    SubTableTerm [ UnidiffSpec  SubTableTerm ] ...
TableTerm ::=
    TableReference [ IntersectSpec  TableReference ] ...
SubTableTerm ::=
    SubTableReference [ IntersectSpec  SubTableReference ] ...
UnidiffSpec ::=
    UnidiffOp  [ CorrespondingSpec ]
```

```
UnidiffOp ::=
    UNION [ALL] | DIFF | EXCEPT
IntersectSpec ::=
    INTERSECT  [ CorrespondingSpec ]
CorrespondingSpec ::=
    CORRESPONDING  [ BY (FieldNameList) ]
```

Explanation: UNION computes the set theoretical union of both input sets. If ALL is specified then duplicate tuples are retained otherwise they are removed. DIFF and INTERSECT compute the set theoretical difference and intersection, resp. Duplicate tuples are removed.

EXCEPT is a synonym for DIFF.

Note that according to the grammar rules, INTERSECT binds stronger than UNION and DIFF. Associativity is from left to right (see also Precedence of Operators).

The expression

A setop CORRESPONDING BY (C1, ..., Cn) B

where setop is one of the set operators is equivalent to

(SELECT C1, ..., Cn FROM A) setop (SELECT C1, ..., Cn FROM B)

The expression

A setop CORRESPONDING B

is equivalent to

A setop CORRESPONDING BY (C1,..,Cn) B

where C1, ..., Cn are the fields with common names in A and B in the order of A. If A and B have no fields in common, an error is returned.

The result types of DIFF are those of the left operand. With UNION and INTERSECT, the type adaption rules (see Data Types and Type Compatibility) are applied to determine the result types.

DIFF preserves the naming of its left operand, i.e. if the i-th field of the left operand of DIFF is named 'xyz' (or is unnamed), then the i-th field of the result of DIFF is also named 'xyz' (is unnamed, resp.).

The i-th result field of a UNION or INTERSECT with operands A and B is unnamed if the i-th field of either A or B is unnamed or if their names differ, otherwise it is named and has the common input name as name.

Updatability: A *TableExpression* is updatable if no UNION, INTERSECT, DIFF is specified and if the underlying *TableReference* is updatable.

CorrelationNames: A *TableExpression* exports no *CorrelationName* if one of the set operators UNION, INTERSECT, DIFF is specified, otherwise it exports the *CorrelationName* of the constituting *TableReference*.

Example:

```
SELECT * FROM quotations
UNION CORRESPONDING BY (suppno)
SELECT * FROM suppliers
```

5.12 TableReference, SubTableReference

A *TableReference* or *SubTableReference* is the constituent of the FROM clause and of UNION / INTERSECTION / DIFF expressions (*TableExpression*s).

Syntax:

```
TableReference ::=
    [ TABLE ] TableSpec [ Alias ]
  | FUNCTION TableFunction
  | SelectExpression
  | ( TableExpression ) [ Alias ]
  | JoinedTable
  | FULLTEXT SpecialFulltextTable

SubTableReference ::=
    SelectExpression
  | ( SubTableExpression )

TableSpec ::=
    LocalTableName
  | RemoteTableName

LocalTableName ::=
    TableName

RemoteTableName ::=
    TableName@Dbname[@Hostname[:Port]]
```

```
Alias ::=
    [ CorrelationName ] [ (FieldNameList) ]

FieldNameList ::=
    FieldName [, FieldName ] ...

TableFunction ::=
    FunctionName ( ExpressionList )

ExpressionList ::=
    Expression [ , Expression ] ...
```

Explanation: *TableReference* and *SubTableReference* are the building blocks for *TableExpression* and *SubTableExpression*.

Note: *TableExpression* constitutes the top query block and subqueries in FROM clause, *SubTableExpression* constitutes subqueries in SELECT, WHERE, HAVING clause.

Hint: If a TableExpression TE is needed on a place where only a SubTableExpression is allowed, then the equivalent expression `( SELECT * FROM (TE))` can be used. As can be seen from the grammar, this is a SubTableReference and thus also a SubTableExpression.

If *Alias* is specified with a FieldNameList, the exported field names are defined by the specified FieldNames. The number of FieldNames specified must be equal to the arity of *TableName* or *TableExpression*, resp.

If no *FieldNameList* is specified, the exported field names are derived from the underlying syntactic construct (e.g. *Tablename* exports the field names of the fields of the specified table).

Updatability: A *TableReference* is updatable if one of the following holds:

- A TableName T is specified and T is a basetable or an updatable view.
- An updatable SelectExpression is specified.
- An updatable (TableExpression) is specified.

CorrelationNames: see TableReferences, CorrelationNames and Scopes.

RemoteTableName: RemoteTableNames and all other remote identifiers must be specified in accordance to local and remote case sensitivity settings. Database names are always treated case sensitive. Host names are always treated case insensitive. If a table residing in a case sensitive database is referenced from a case insensitive databases, then the database name, table name and column names must be written case sensitive and surrounded by double quotes, e.g.

```
SELECT "SomeThing" FROM "TableName"@case_sens_db@host
```

If a table residing in a case insensitive database is referenced from a case sensitive databases, then the table and column names must be written in uppercase letters, e.g.

```
SELECT SOMETHING FROM TABLENAME@case_insens_db@host
```

Note also that the current user requires access privileges using the same credentials as for the local and remote database. If case insensitie databases are involved, make sure that usernames are written in uppercase letters, otherwise they will not match on case insensitive databases.

Example: In the following example, an alias is used for each of the *TableReference*s, the first consisting of a *CorrelationName* only, the second with a field list.

```
SELECT  q.partno, supp.sno, supp.addr, supp.name
FROM quotations q, suppliers supp (sno, name,addr)
WHERE q.suppno = supp.sno
```

Example: The following example needs an Alias for its subquery to reference its result field - it works with or without a CorrelationName. Both solutions are shown:

```
SELECT AVG (q.cnt) FROM
   (SELECT COUNT (*)
   FROM quotations
   GROUP BY partno) q (cnt)

SELECT AVG(cnt) FROM
    (SELECT COUNT (*)
    FROM quotations
    GROUP BY partno)  (cnt)
```

Example: The following example is a distributed join using a remote database otherdb@server5.

```
SELECT  q.partno, supp.sno
FROM quotations q, suppliers@otherdb@server5 supp
WHERE q.suppno = supp.sno
```

5.12.1 TableFunction

A TableFunction may appear in the FROM clause of a SELECT statement. It is a built-in or user-defined procedure and acts like a basetable or view in the surrounding SQL statement. The fields of the result tuples may be treated any like other basetable fields in the WHERE, GROUP BY, SELECT and ORDER BY clause.

For details on adding further user-defined Table Functions see stoproc.pdf.

Currently there are two built-in table functions available. First, the Java based JDBCReader offers read-only access to any reachable JDBC data source. The built-in JDBCReader requires a Java runtime environment to be configured for this database. The Transbase JDBCReader is called with:

```
SELECT * FROM FUNCTION JDBCReader(
  'jdbc:transbase://hostname:2024/dbname',
  'user','passwd','select * from sometable')
```

The following steps show the necessary configuration for using third-party JDBC drivers to be used by the JDBCReader.

1. Add the third-party driver to the JRE's CLASSPATH. Make sure that the file is accessible for the Transbase service. Note that the CLASSPATH points to the JAR file, not only to the directory:

```
ALTER EXTERNAL OPTION CLASSPATH "/usr/lib/java/acmesql.jar"
```

2. Make sure the driver registers with the system's JDBC driver manager by providing the driver's fully qualified class name

```
ALTER EXTERNAL OPTION JAVA "-Djdbc.drivers=com.acme.jdbc.Driver"
```

3. Allow the driver to access network resources. Possibly other permissions are also required:

```
ALTER EXTERNAL OPTION JAVAPERMISSIONS "java.net.NetPermission"
```

Now the third-party database may be accessed by calling the JDBCReader using the appropriate connection URL.

In contrast to the generic JDBCReader, the native OraReader offers a high performance read-only access to Oracle databases.

```
SELECT * FROM FUNCTION OraReader(
  '//host:[port][/service name]',
  'user','passwd','select * from oraTable')
```

The OraReader requires an OCI client software installed on the machine where the Tranbase engine is running. Because of its ease of handling, the Oracle Instant client is recommenced over the standard Oracle client installation for this purpose. The software is freely available from Oracle. Please follow the installation instructions and make sure that PATH (Windows) or LD_LIBRARY_PATH (Unix platforms) environment variables are set to include the Instant Client, before the Transbase service is started. Please consult the Oracle documentation on the environment variables ORACLE_HOME, TNS_ADMIN et cetera, if you are planning to use the standard Oracle client and connect via Oracle TNS services. Note that these environment variables also must be set before the Transbase service is started.

These built-in table functions are provided as powerful data import facilities. Their input parameters consist of a connection string, username and password for opening a connection to the data source. Note that all parameters are normal StringLiterals and the escape syntax conventions apply. These are followed by arbitrary SQL statements to be processed by the database engine of the data source. The result is automatically mapped to Transbase data types where possible. If a datatype (e.g. Oracle's LONG and LONG RAW) cannot be mapped, try to cast it to a mappable type on the remote system. In the next step the data is processed according to an arbitrary Transbase SQL that encloses the table function. In particular, this Transbase SQL statement may be an INSERT or SPOOL statement.

5.13 JoinedTable (Survey)

A *JoinedTable* combines tables with an explicit join operator.

Syntax:

```
JoinedTable ::=
    TableReference CROSS JOIN TableReference
  | TableReference UNION JOIN TableReference
```

```
  | TableReference NATURAL Joinop TableReference
  | TableReference [ Jointype ] JOIN TableReference [ Joinpred ]
  | ( JoinedTable )

Jointype ::=
    INNER
  | LEFT   [ OUTER ]
  | RIGHT  [ OUTER ]
  | FULL   [ OUTER ]

Joinpred ::=
    ON SearchCondition
  | USING ( FieldNameList )
```

Explanation: CROSS JOIN is a syntactic variant for Cartesian product, i.e. the following expressions are semantically equivalent:

```
A CROSS JOIN B
SELECT * FROM A,B
```

The expression `A UNION JOIN B` (where A has a fields and B has b fields) is semantically equivalent to :

```
SELECT  A.*, NULL, NULL, ...   -- b NULLs
  FROM A
UNION
SELECT  NULL, NULL, ..., B.*   -- a NULLs
  FROM B
```

The result table has a+b fields and each tuple either has the first a or the last b fields all NULL.

The other join variants are described in the following chapters.

CorrelationNames: A JoinedTable exports the *CorrelationName*s of both participating *TableReference*s - i.e. none, one or both CorrelationNames.

Example:

```
SELECT A.*, B.*
FROM  A UNION JOIN B.
```

5.13.1 JoinedTable with ON Clause and USING Clause

This chapter discusses the *JoinedTable* with *Jointype* [INNER] JOIN.

Syntax:

```
JoinedTable ::=
    TableReference [ INNER ] JOIN TableReference
    ON SearchCondition
  | TableReference [ INNER ] JOIN TableReference
    USING ( FieldNameList )
```

Explanation:

Case (1), ON Clause specified: Let searchcond be the search condition. The expression

```
A [INNER] JOIN B ON searchcond
```

semantically is equivalent to

```
SELECT * FROM A, B WHERE searchcond
```

Case (2), USING Clause specified: If the USING Clause is specified then let C1, ..., Cn denote the *FieldNameList*. All Ci's must be fields of both A and B. The expression

```
A [INNER] JOIN B USING (C1, ... ,Cn)
```

semantically is equivalent to

```
SELECT A.C1, A.C2, ... , A.Cn,
       <other A fields>, <other B fields>
FROM A, B
WHERE  A.C1=B.C1 AND ... AND A.Cn=B.Cn
```

Each of the result fields Ci appears only once in the result table (i.e. the number of result fields is $a + b - n$).

The result fields C1, ..., Cn have no CorrelationNames (even if the constituting TableReferences A and B export CorrelationNames, say "a" and "b"). Thus, in the surrounding *SelectExpression*, C1, ..., Cn can only be referenced by their unqualified name. The remaining fields of A and B have CorrelationNames "a" and "b" of A and B (if they exist). Note that also `a.*` and `b.*` refer to the remaining fields in their original order without any Ci.

5.13.2 JoinedTable with NATURAL

This chapter discusses the *JoinedTable* with *Jointype* NATURAL [INNER] JOIN.

Syntax:

```
JoinedTable ::=
    TableReference NATURAL [ INNER ] JOIN TableReference
```

Explanation: The expression

```
A NATURAL [INNER] JOIN B
```

is equivalent to

```
A [INNER] JOIN B USING (C1,..Cn)
```

where C1, ..., Cn are the fields (in the order of A) which are common to A and B.

If no fields are common, the expression degenerates to `A UNION JOIN B`.

Example: The following statements all deliver the same result:

1.
```
SELECT  q.partno, q.suppno, q.price, q.delivery_time,
    q.qonorder, i.description, i.qonhand
FROM quotations q, inventory i
WHERE q.partno= i.partno
```

2.
```
SELECT q.partno, q.suppno, q.price, q.delivery_time,
       q.qonorder, i.description, i.qonhand
FROM quotations q  JOIN  inventory i
ON q.partno= i.partno
```

3.
```
SELECT  q.partno, q.*, i.*
FROM quotations q  JOIN  inventory i USING (partno)
```

4.
```
SELECT  q.partno, q.*, i.*
FROM quotations q  NATURAL JOIN  inventory i
```

Note the meaning of q.* and i.* in the context of USING and NATURAL. Note also that suppno and partno are opposite to their original order in quotations.

5.13.3 JoinedTable with OUTER JOIN

This chapter discusses the OUTER JOIN variants of the join operators.

Syntax:

```
JoinedTable ::=
    TableReference [ Jointype ] JOIN TableReference [ Joinpred ]
  | TableReference NATURAL Joinop TableReference
  | ( JoinedTable )

Jointype ::=
    LEFT    [ OUTER ]
  | RIGHT   [ OUTER ]
  | FULL    [ OUTER ]

Joinpred ::=
    ON SearchCondition
  | USING ( FieldNameList )
```

Explanation: Specification of OUTER has no effect

We discuss the join variants

Case (1), ON Clause specified: Assume the expressions LJ, RJ, FJ, IJ as:

LJ	A LEFT JOIN B ON searchcond
RJ	A RIGHT JOIN B ON searchcond
FJ	A FULL JOIN B ON searchcond
IJ	A INNER JOIN B ON searchcond

Let *innerjoin* denote the result of IJ. Then the result sets of LJ, RJ, FJ are defined as:

Result LJ	innerjoin UNION ALL leftouter
Result RJ	innerjoin UNION ALL rightouter
Result FJ	innerjoin UNION ALL fullouter

where *leftouter* , *rightouter*, *fullouter* are defined as follows:

***leftouter*:** the set of all tuples a from A which do not participate in innerjoin, extended to the right with NULL values up to the arity of innerjoin.

***rightouter*:** set of all tuples b from B which do not participate in the set innerjoin, extended to the left with NULL values up to the arity of innerjoin.

***fullouter*:** leftouter UNION ALL rightouter.

Case (2), USING Clause specified: Let JU denote the join expression

```
A lrf JOIN B USING (C1, ... ,Cn)
```

where lrf is one of LEFT, RIGHT, FULL.

Let searchcond be the following search condition:

```
A.C1=B.C1 AND ... AND A.Cn=B.Cn
```

Then the result of JU is defined to be equivalent to:

```
SELECT  COALESCE(A.C1,B.C1), ..., COALESCE(A.Cn,B.Cn),
    <other fields of A> , <other fields of B>
FROM A lrf JOIN B ON searchcond
```

The USING variant works like the ON variant except that the specified common fields appear only once in the result (always the not-NULL part of the field if any appears).

Note that the COALESCEd fields do not have a *CorrelationName* and that the CorrelationNames exported by A and B do not include the COALESCEd fields.

Case (3), NATURAL specified: Let NJ denote the expression:

```
A NATURAL lrf JOIN B
```

where lrf is one of LEFT, RIGHT, FULL.

NJ is equivalent to

```
A lrf JOIN B ON (C1, ..., Cn)
```

where C1, . . . , Cn are all fields with *identical* names in A and B (in the order as they appear in A).

Example: For all following examples, we assume excerpts S and Q from suppliers and quotations as shown in tables 5.4 and 5.5.

S	
suppno	name
51	DEFECTO PARTS
52	VESUVIUS,INC
53	ATLANTIS CO.

Table 5.4: Table S

Q	
suppno	partno
50	221
51	221
53	222
53	232

Table 5.5: Table Q

Example:

```
SELECT * FROM S LEFT JOIN Q ON S.suppno = Q.suppno
```

S.suppno	S.name	Q.suppno	Q.partno
51	DEFECTO PARTS	51	221
52	VESUVIUS INC	NULL	NULL
53	ATLANTIS CO	53	222
53	ATLANTIS CO	53	232

Example:

```
SELECT * FROM S RIGHT  JOIN Q ON S.suppno = Q.suppno
```

S.suppno	S.name	Q.suppno	Q.partno
NULL	NULL	50	221
51	DEFECTO PARTS	51	221
53	ATLANTIS CO	53	222
53	ATLANTIS CO	53	232

Example:

```
SELECT * FROM S FULL  JOIN Q ON S.suppno = Q.suppno
```

S.suppno	S.name	Q.suppno	Q.partno
NULL	NULL	50	221
51	DEFECTO PARTS	51	221
52	VESUVIUS INC	NULL	NULL
53	ATLANTIS CO	53	222
53	ATLANTIS CO	53	232

Example:

```
SELECT * FROM S LEFT  JOIN Q USING (suppno)
```

suppno	S.name	Q.partno
51	DEFECTO PARTS	221
52	VESUVIUS INC	NULL
53	ATLANTIS CO	222
53	ATLANTIS CO	232

Note that the first result field can only be referenced by the unqualified name `suppno`. Neither `S.*` nor `Q.*` include the `suppno` field.

Example:

```
SELECT * FROM S NATURAL FULL JOIN Q
```

suppno	S.name	Q.partno
50	NULL	221
51	DEFECTO PARTS	221
52	VESUVIUS INC	NULL
53	ATLANTIS CO	222
53	ATLANTIS CO	232

Note that the first result field can only be referenced by the unqualified name `suppno`. Neither `S.*` nor `Q.*` include the `suppno` field.

5.14 TableReferences, CorrelationNames and Scopes

CorrelationNames may be used in qualified field references - they are of the form "q.field" where q is a CorrelationName.

A *TableReference* which constitutes a FROM clause operand exports a CorrelationName in the following cases:

- If the TableReference specifies an *Alias*, then the *CorrelationName* specified in the *Alias* is exported.

- If the TableReference is a *TableName* without a specified *Alias*, then a *CorrelationName* which is identical to the *TableName* implicitly is exported.
- If the *TableReference* is a *JoinedTable* then the CorrelationName(s) exported by *JoinedTable* are exported (see JoinedTable).

The Scope of the *CorrelationName* is the *SelectExpression* that immediately contains the *TableReference*. However, excluded are SelectExpressions which are nested in the containing one and define a *CorrelationName* with the same name.

Example: In the following example, TableReferences "quotations" and "suppliers s" export CorrelationNames "quotations" and "s", resp.:

```
SELECT  quotations.*, s.name
FROM quotations, suppliers s
WHERE quotations.suppno = s.suppno
```

Example: In the following example, the JoinedTable exports CorrelationNames "q" and "s":

```
SELECT s.name, q.price
FROM quotations q JOIN suppliers s ON q.suppno = s.suppno
```

Example: In the following example, the JoinedTable also exports CorrelationNames "q" and "s", but the common result field "suppno" has no CorrelationName and q. and s. do not include the field "suppno":

```
SELECT suppno, s.*, q.*
FROM  quotations q NATURAL JOIN suppliers s
```

5.15 SelectStatement

The *SelectStatement* is the top level construct of TB/SQL to retrieve tuples.

Syntax:

```
SelectStatement ::=
    TableExpression { [SortSpec] | [UpdSpec] }

SortSpec ::=
```

```
    ORDER BY SortElem [, SortElem ] ...
  | ORDER BY ALL

SortElem ::=
    { FieldName | IntegerLiteral } [ ASC | DESC ]

UpdSpec ::=
    FOR UPDATE
```

Explanation: The ORDER-BY-clause sorts the result tuples of the *TableExpression*. If more than one *SortElem* is specified, a multi-field sort is performed with the first *SortElem* being the highest sort weight etc.

If a *SortElem* is given via an IntegerLiteral i, it refers to the i-th result field of the *TableExpression*, otherwise there must be an identically named result field of the TableExpression and the SortElem refers to that field. Field numbering starts at 1. The next example below shows two equivalent sort specifications.

By default the sort order is ascending unless explicitly descending order is required ('DESC').

Note: For sorting result fields of character types (VARCHAR / CHAR / BINCHAR), TBX application programs and ESQL programs can specify sortorders which override the machine code order (see chapter 2.3 (User Defined Sortorder) and special manuals on application programming).

A *SelectStatement* is updatable if no ORDER BY-clause is specified and if the *TableExpression* itself is updatable (see TableExpression).

A *SelectStatement* is called a SELECT FOR UPDATE query if the *UpdSpec* is specified. An *UpdSpec* is only allowed in an TBX or ESQL application program (see Manuals TBX and TB/ESQL). It is necessary if and only if a subsequent UPDPOS or DELPOS statement (Update- or Delete-Positioned) is intended against the query. In case of an *UpdSpec*, the *SelectStatement* must be updatable.

Note: There is no predictable ordering of result tuples, if no order-by-clause is specified.

Privileges: The current user must have SELECT privilege on each table (base table or view) specified in any FROM-clause of any QueryBlock which occurs in the SelectStatement.

Locks: All tables and views referenced in the *SelectStatement* are automatically read locked.

If the *UpdSpec* is given, the (one and only) table in the outermost *QueryBlock* is update locked.

Example:

```
SELECT partno, price FROM quotations ORDER BY partno

SELECT partno, price FROM quotations ORDER BY 1

SELECT * FROM quotations FOR UPDATE
```

5.16 InsertStatement

The *InsertStatement* inserts one or several constant tuples or a computed set of tuples into a table or updatable view.

Syntax:

```
InsertStatement ::=
    INSERT INTO TableSpec [ (FieldNameList )]
    Source

TableSpec ::=
    LocalTableName
  | RemoteTableName

FieldNameList ::=
    FieldName [, FieldName ] ...

Source ::=
    VALUES  (ValueList)
  | TABLE ( (ValueList) [, (ValueList) ] ... )
  | TableExpression
  | DEFAULT  VALUES

ValueList ::=
    Expression [, Expression ] ...
```

Explanation: The table specified by *TableName* must be updatable (i.e. a base table or an updatable view.

All *FieldNames* in a specified *FieldNameList* must be unique and must be fields of the specified table.

If no *FieldNameList* is specified then there is an implicitly specified *FieldNameList* with all fields of the specified table in the order of the *FieldNames* of the corresponding *CreateTableStatement* or *CreateViewStatement*, resp.

The number of *Expressions* in a *ValueList* or the number of fields of the result of the *TableExpression* must be the same as the number of *FieldNames* and the corresponding types must be compatible.

Each *ValueList* or each result tuple of the *TableExpression*, resp., represents a tuple which is inserted into the table.

If DEFAULT VALUES is specified, then a tuple consisting of the default value of each field is inserted.

For each tuple t to be inserted, the i-th *FieldName* in the *FieldNameList* specifies to which table field the i-th field of t is assigned. The null value is assigned to all fields of the table which are not specified in the FieldNameList. Null values also can be specified explicitly by the constant NULL in ValueList.

If the specified table is a view, then insertion is effectively made into the underlying base table and all fields of the base table which are not in the view are filled up with null values.

For a view v where the WITH CHECK OPTION is specified the insertion of a tuple t fails if t does not fulfill the *SearchCondition* of the view definition of v or any other view on which v is based.

The *InsertStatement* fails if a NULLConstraint or a key constraint or a UNIQUE constraint (defined by a CREATE UNIQUE INDEX ...) would be violated or if a type exception occurs (see DataType and Type exception).

Privileges: The current user must have INSERT privilege on the specified table.

If a *TableExpression* is specified, the user must additionally have the corresponding SELECT-privileges as if the *TableExpression* were run as a *SelectStatement* (see SelectStatement).

Locks: The table referenced by *TableName* is update locked automatically.

Example:

```
INSERT INTO suppliers
VALUES (80, 'TAS', 'Munich')
```

```
INSERT INTO suppliers (name,suppno)
VALUES
('Smith & Co', (SELECT MAX (suppno)+1 FROM suppliers) )

INSERT INTO suppliers TABLE
(
   (81,'xy','ab'),
   (82,'yz','bc')
)

INSERT INTO suppliers@otherdb@server5 (name,suppno)
VALUES ('Smith & Co',
       (SELECT MAX (suppno)+1 FROM
               suppliers@otherdb@server5) )

INSERT INTO suppliers
SELECT * FROM suppliers2
```

Note: The TABLE variant of the Insert Statement (several constant tuples) semantically is processed via a UNION ALL of the specified tuples. This has an undesirable side effect if the target table has a CHAR(*) or BINCHAR(*) field: if constants of different length occur, they are type adapted according to the UNION rules (blank padding) before they are inserted into the target table.

5.17 DeleteStatement

The DeleteStatement deletes tuples from a table or an updatable view.

Syntax:

```
DeleteStatement ::=
    DELETE FROM TableSpec [CorrelationName]
    [ WHERE SearchCondition ]
```

TableSpec ::= LocalTableName — RemoteTableName

Explanation: The table specified by *TableName* must be updatable (i.e. a base table or an updatable view).

All tuples from the specified table which satisfy the *SearchCondition* are deleted.

If the *SearchCondition* is omitted, all tuples from the specified table are deleted.

If tuples are deleted from an updatable view, the deletion is made on the underlying base table.

Note: It is allowed to refer to the table to be modified in a subquery of the *DeleteStatement* (in the *SearchCondition*). See the examples below and also General Rule for Update.

Deletion of many tuples may be slow if secondary indexes exist. However, deletion of all tuples in a table (WHERE clause is omitted) is very fast.

Privileges: The current user must have DELETE-privilege on the specified table.

If TableExpressions occur in the *SearchCondition*, the user must additionally have the corresponding SELECT-privileges as if the TableExpressions were run as SelectStatements (see SelectStatement).

Locks: The table referenced by *TableName* is update locked automatically.

If a remote table is specified as the target of the DELETE operation, all subqueries (if any) must specify tables residing on the same database. However, if the target table is local, any tables (remote or local) may be specified in subqueries.

Example:

```
DELETE FROM quotations

DELETE FROM suppliers
WHERE suppno = 70

DELETE FROM suppliers
WHERE suppno =
    (SELECT MAX (suppno)
    FROM suppliers@otherdb@server5)

DELETE FROM suppliers@otherdb@server5 s
WHERE NOT EXISTS
    (SELECT *
    FROM quotations@otherdb@server5
    WHERE suppno = s.suppno)
```

See also General Rule for Updates.

5.18 UpdateStatement

The *UpdateStatement* updates a set of tuples in a table or an updatable view.

Syntax:

```
UpdateStatement ::=
UPDATE TableSpec [CorrelationName]
    SET AssignList
    [ WHERE {SearchCondition | CURRENT} ]

TableSpec ::=
    LocalTableName
  | RemoteTableName

AssignList ::=
    Assignment [, Assignment ] ...

Assignment ::=
    FieldName = Expression
```

Explanation: All FieldNames on the left hand side of the Assignments must be unique and must be fields of table *TableName*.

The user must have UPDATE privilege on all these fields and the table must be updatable.

The types of the fields must be compatible with the types of the corresponding Expressions.

The effect of the UpdateStatement is that all tuples of the specified table which fulfill the *SearchCondition* are updated. If no *SearchCondition* is specified then *all* tuples of the table are updated. For each tuple to be updated the fields on the left hand sides of the *Assignments* are updated to the value of the corresponding *Expression* on the right hand side. Unspecified fields remain unchanged.

If the specified table is a view then the update is effectively made on the underlying base table and all fields of the base table which are not in the view remain unchanged.

For a view v where the WITH CHECK OPTION is specified the update of a tuple t fails if the updated tuple would not fulfill the *SearchCondition* of the view definition of v or any other view on which v is based.

The *UpdateStatement* fails if a NULL-Constraint or a key constraint or a UNIQUE constraint (defined by a CREATE UNIQUE INDEX ...) would be violated or if a type exception occurs (see DataType and Type exceptions).

The specification `OF CURRENT` is only allowed at the Programming Interface TBX for an `UPDPOS` call (see TBX Manual).

Note: It is allowed to update key fields.

Update of key fields runs considerably slower than update of non-key fields only. See also Note in CreateTableStatement.

Note: It is allowed to refer to the table to be updated in a subquery of the UpdateStatement (in the AssignList or SearchCondition). See the example and also General Rule for Update.

Privileges: The current user must have UPDATE-privilege on all fields specified on the left hand sides of the *Assignments*.

If there are *TableExpressions* on the right hand side of the *Assignments* or in the *SearchCondition*, the user additionally must have corresponding SELECT-privileges as if the TableExpressions were run as SelectStatements (see Select-Statement).

Locks: The table referenced by *TableName* is update locked automatically.

If a remote table is specified as the target of the UPDATE operation, all subqueries (if any) must specify tables residing on the same database. However, if the target table is local, any tables (remote or local) may be specified in subqueries.

Example:

```
UPDATE quotations
SET price = price * 1.1, delivery_time = 10
WHERE suppno = 53 AND partno = 222

UPDATE quotations@otherdb@server5 q
SET price = price * 1.1
WHERE price =
    (SELECT MIN (price)
    FROM quotations@otherdb@server5
    WHERE suppno = q.suppno)
```

5.19 MergeStatement

The *MergeStatement* serves as a combination of the *InsertStatement* with the *UpdateStatement*. It combines the effects of both these statements within a single one.

Syntax:

```
MergeStatement ::=
    MERGE INTO TargetTable
    USING SourceTable ON (JoinPredicate)
    MatchClause  NonMatchClause

TargetTable ::=
    TableSpec

SourceTable ::=
    TableExpression

JoinPredicate ::=
    SearchCondition

MatchClause ::=
    WHEN MATCHED THEN UPDATE SET AssignList

NonMatchClause ::=
    WHEN NOT MATCHED THEN
      INSERT [ (FieldNameList) ] VALUES ValueList

ValueList ::=
    Expression [, Expression ] ...
```

Explanation: The table specified by TargetTable must be updatable, SourceTable may be an expression delivering a set of tuples or a base table name. The JoinPredicate refers to fields of TargetTable and SourceTable. It can be thought of as a loop executed on the SourceTable S - for each tuple of S there must be either no tuple in TargetTable which matches the predicate or exactly one tuple in TargetTable which matches. In the first case, the NonMatchClause is executed which inserts fields of the current tuple of S into the TargetTable. In the second case, the MatchClause is executed which updates fields of the TargetTable with field values of the current tuple of S.

Thus, in the NonMatchClause, the FieldNameList must refer to field names of TargetTable only and the ValueList must refer to fields of the SourceTable only.

The current user must have INSERT and UPDATE privilege on TargetTable.

Locks: The TargetTable is update locked automatically.

Example:

```
MERGE INTO suppliers tar
USING (SELECT * FROM newsuppliers) src
ON (tar.suppno = src.suppno)
WHEN MATCHED THEN UPDATE SET tar.address = src.address
WHEN NOT MATCHED THEN INSERT VALUES(suppno, name, address)
```

5.20 General Rule for Updates

The semantics of all modification operations (INSERT, UPDATE, DELETE) is that of a *deferred update*, i.e. conceptually the modification is performed in two phases:

1. Compute the whole modification information (tuples to be inserted, tuples to be changed and their values to replace existing tuples, tuples to be deleted, resp.). In this phase the table to be modified remains unchanged.
2. Execute the modification.

This principle allows to specify the modification referring to the old state of the table and defines the modification as an atomic step.

5.21 Rules of Resolution

For the semantics of nested *QueryBlocks* (*SelectExpressions*) it is necessary that any *Field* and any *SetFunction* can be resolved against exactly one *QueryBlock*. This block is then called the resolution block of the Field or the SetFunction, resp.

5.21.1 Resolution of Fields

An unqualified Field `fld` (see FieldReference) has as its resolution block the innermost surrounding QueryBlock q whose FROM-clause contains a *TableReference* which exports a field named `fld`. If there is more than one such *TableReference* in q, the resolution fails and an error is returned.

A qualified Field `r.fld` has as its resolution block the innermost surrounding QueryBlock q whose FROM-clause contains a *TableName* or *CorrelationName* `r`. If the corresponding *TableReference* does not export a field named `fld`, the resolution fails and an error is returned.

5.21.2 Resolution of SetFunctions

In most cases, the resolution block of a *SetFunction* is the innermost surrounding *QueryBlock*.

Example:

```
SELECT partno
FROM quotations
GROUP BY partno
HAVING
    MAX(price) - MIN(price)
    >
    AVG(price) * 0.5
```

The resolution of all three SetFunctions is the only (and innermost) QueryBlock. Thus, they are computed for each group of parts.

In general, the resolution of `count(*)` is always the innermost surrounding QueryBlock; for all other forms of SetFunctions, the resolution block is the innermost resolving QueryBlock over all fields inside the SetFunction.

Example:

```
SELECT partno
FROM quotations q1
GROUP BY partno
HAVING
    (SELECT COUNT(*) FROM quotations q2
    WHERE q2.partno = q1.partno AND
    q2.price = MIN (q1.price))
    > 1
```

Here, `COUNT(*)` refers to the inner *QueryBlock* whereas `MIN(q1.price)` refers to the outer *QueryBlock* and thus computes as the minimum price over the current group of parts.

Arbitrary (single-valued) Expressions are allowed as arguments of SetFunctions. It is even allowed to nest SetFunctions as long as the resolution block of the inner SetFunction surrounds the resolution block of the outer SetFunction.

For each SetFunction s with resolution block q, s must not appear in the WHERE-clause of q.

Chapter 6

Load and Unload Statements

In addition to the TB/SQL language, the TransbaseCD Retrieval System offers language constructs to control the data transfer from CD ROM to the disk cache and to unload the disk cache. These constructs are called *LoadStatement* and *UnloadStatement*, resp. These statements are only relevant and valid in CD-ROM databases and are explained in the Transbase CD-ROM Database Guide.

Chapter 7

Tbmode Statements

TBMODE Statements serve to configure runtime parameters of a Transbase kernel. They are not subject to the transaction concept; i.e. they can be issued inside or outside a transaction.

Usage of TBMODE statements:

- In interactive applications like UFI and TBI they are typed like any other SQL statement.
- At the programming interface ESQL they are run with EXEC SQL like other SQL statement.
- At the programming interface TBX they are issued via a TbxTbmode(...) call.

The effect of a TBMODE statement is restricted to the application which issues it.

One class of TBMODE Statements are for tuning purposes. They configure buffer sizes for the catalog manager and for tuple transport between kernel and application. They are explained in chapter 7.1 Tbmode Tuning Statements.

Another class influences the behaviour of a kernel with respect to opening and closing of the database files. This is of importance for maintaining a database on several files which are distributed e.g. on a jukebox of magnetooptical disks (MO jukebox). These statements are explained in chapter 7.2 Tbmode File Statements.

A third class influences the locking behaviour with respect to lock granularity, i.e. one can switch between page locks, table locks and automatic choice by Transbase.

7.1 Tbmode Tuning Statements

7.1.1 TbmodeCatalogStatement

This statement serves to configure the size of the main memory catalog buffers.

Syntax:

```
TbmodeCatalogStatement ::=
    TBMODE CATALOG [ BUFFER ] CatbufConfig

CatbufConfig ::=
    DEFAULT
  | ListofSizes
  | SIZE KBytes

ListofSizes ::=
    SYSBLOB  Elements ,
    SYSCOLUMN  Elements ,
    SYSCOLUMNPRIV  Elements ,
    SYSINDEX  Elements ,
    SYSTABLE  Elements ,
    SYSTABLEPRIV  Elements ,
    SYSVIEW Elements
```

Any permutation of the seven tables is allowed

```
KBytes ::= IntegerLiteral
Elements ::= IntegerLiteral
```

Explanation: If a *ListofSize* is specified, then for each catalog table a buffer is allocated that can hold as many tuples as is specified by Elements. A number of 0 in Elements is legal. Note that a missing table is assumed to have been specified with 0. Note further that there are two more catalog tables (SYSUSER and SYSVIEWDEP) which are never buffered - a specification is legal but is ignored.

If a SIZE is specified then the number is interpreted as KBytes and is the total size of all catalog buffers. Internally Transbase splits the specified memory pool into pieces for the single catalog tables according to some default characteristics.

If DEFAULT is specified then this is equivalent as if a SIZE of 25 KB had been specified. This is the mode that a kernel starts when forked for an application.

The catalog buffer influences the performance of query compilation (but not that of evaluation). It is managed on a LRU (least recently used) strategy. A query

compilation occurs with each DDL statement and with each dynamic DML and RUN statement (issued via TBI or UFI or any TBX or ESQL application). For stored queries, the storage process itself needs a compilation whereas the (repeated) execution does not incur a compilation (except in the seldom case that a schema change makes the recompilation of the stored query necessary). Note that the ESQL interface automatically stores all queries which have at least one host variable.

Example: Specify an empty catalog buffer:

```
TBMODE CATALOG SIZE 0
```

Specify a buffer with default size:

```
TBMODE CATALOG DEFAULT
```

Specify a buffer with a given memory size:

```
TBMODE CATALOG SIZE 80
```

Specify a buffer with tailored sizes:

```
TBMODE CATALOG
        SYSCOLUMN  300 ,
        SYSCOLUMNPRIV  10 ,
        SYSINDEX  20 ,
        SYSTABLE  80 ,
        SYSTABLEPRIV  20 ,
```

The last example would be suitable for example if no BLOBs and no Views are in the database or are not addressed by the application.

7.1.2 TbmodeResultBufferStatement

This statement serves to configure the size of the result buffer which collects result tuples before sending them to the application.

Syntax:

```
TbmodeResultBufferStatement ::=
    TBMODE RESULT [ BUFFER ] ResultConfig

ResultConfig ::=
    DEFAULT
  | TUPLES Tuplenumber

Tuplenumber ::= IntegerLiteral
```

Explanation: If DEFAULT is specified then the buffer has a size of 4 KBytes. If a Tuplenumber n (positive Integer) is specified then for each SELECT Query, n result tuples are computed before sending the buffer to the application (which of course transparently fetches the tuples one at a time). Even with large n, the maximum size of the buffer is always 4 KBytes.

Note that the maximum buffer size is recommendable to optimize the total time to process the SELECT query. A small value of n, however, reduces the time until the application receives the first tuple or the first few tuples. However, it can have a negative effect on the time to get all tuples.

Example: Specify a minimum sized result buffer:

```
TBMODE RESULT BUFFER TUPLES 1
```

Reinstall the default size (maximum) size:

```
TBMODE RESULT BUFFER DEFAULT
```

7.1.3 TbmodeSortercacheStatement

This statement serves to configure the size of the main memory buffer for sorting operations (relevant for ORDER BY, GROUP BY, sort merge joins etc.). The configuration comes into effect at the start of the next transaction.

Syntax:

```
TbmodeSortercacheStatement ::=
    TBMODE SORTERCACHE Size [ KB ]

Size ::= IntegerLiteral
```

Explanation: *Size* is the desired size of the sortercache in KB. Big sizes favour the computing time for sorter operations but need more resources for the kernel which processes the application.

The default size of the buffer is in the range of 128 KB.

Note that the feasibility of sorter operations does not depend on the size of the buffer.

Note that the reconfiguration of the sortercache is deferred until the start of the next transaction. For the sake of clarity, it is thus recommended to issue that statement outside of a transaction.

Example:

```
TBMODE SORTERCACHE 1024 KB
```

7.1.4 TbmodeOptimizerStatement

This statement serves to switch the Transbase Query Optimizer between two possible states.

Syntax:

```
TbmodeOptimizerStatement ::=
    TBMODE OPTIMIZER OptimizerConfig

OptimizerConfig ::=
    GLOBAL
  | GREEDY
```

Explanation: As a default, the Transbase Query Optimizer runs in GLOBAL state. This means that the queries are optimized such that the overall time to compute the whole query result is minimized.

In the GREEDY state the optimizer tries to evaluate the query such that few result tuples are produced quickly, but possibly at the expense of the overall time of query evaluation. Note that the GREEDY strategy should run in combination with a small result buffer (see chapter 7.1.2 Tbmode Result Buffer Statement).

Actually, the GREEDY mode only has an effect for queries where one of the following constructs occurs:

- the UNION or UNION ALL clause
- the DISTINCT clause (not inside a set function).

Example: Specify the GREEDY optimizing strategy:

```
TBMODE OPTIMIZER GREEDY
```

Reinstall the (default) GLOBAL strategy:

```
TBMODE OPTIMIZER GLOBAL
```

7.1.5 TbmodeMultithreadStatement

This statement sets the Transbase Query Optimizer to one of four possible levels of parallel query execution. This setting is valid throughout the current session.

Syntax:

```
TbmodeMultithreadStatement ::=
    TBMODE MULTITHREAD MultithreadConfig

MultithreadConfig ::=
    MAX | DETERMINISTIC | MIN | OFF
```

Explanation: The default multithread mode for a database is `MULTITHREAD OFF`. This global setting can be controlled with the `tbadmin` tool via options that are similar to those decribed here.

Parallelization is implemented in one singular operator node called ASYNC. This operator provides a tuple buffer and a new thread execution. The thread executes the operator tree portion below the ASYNC node and delivers its result tuples into the buffer. The upper part of the operator tree runs independently in the parent thread and simultaneously retrieves the tuples from the buffer. Access to the buffer is synchronized. The upper thread blocks until tuples are available, the lower thread blocks when the buffer is full. When the lower part of the operator tree is exhausted, i.e. last tuple was delivered, the thread terminates.

Note: Without multithreading Transbase guarantees that data is always processed and returned in the same deterministic sort order, even if no `ORDER BY` was specified. I.e. the same query produced always the same result in the same order. The SQL specification does not demand any reproducible sort order if no `ORDER BY` is used. With multithreaded query processing switched to `MAX` it now likely that data is processed out-of-order. Thus a query will return the same result but possibly in different order if no `ORDER BY` is specified. Only the specification of an `ORDER BY` guarantees a result sort order.

For supporting legacy applications while still enabling multithreaded query execution Transbase supports multithreading in `DETERMINISTIC` mode. This mode offers a fair degree of parallelism while conserving the behavior of single threaded query evaluation.

- `MAX` activates the full potential of multithreading; it establishes data pipelines in query plans that run in parallel, also using out-of-order execution, for improved overall performance.
- `DETERMINISTIC` offers fair parallelism while producing result sets in deterministic output order. Performance is likely to suffer somewhat compared to `MAX`imum parallelism, as data pipelines operate only in first-in, first out mode.
- `MIN` is a rather defensive strategy of parallel query execution; parallel execution is limited to I/O relevant nodes (e.g. REL or REMOTE) and activates work-ahead for the entire `SELECT` query,
- `OFF` means no parallelization at all (default),

Example: Setting Transbase for maximum parallalism:

```
TBMODE MULTITHREAD MAX
```

7.2 Tbmode File Statement

Transbase provides a great flexibility concerning the placement of the files containing the data of the database (diskfiles `tbdsk001` etc.). For example, some diskfiles (perhaps dedicated for BLOBs, see System Guide) can be placed on different disks of a magnetooptical (MO) or even CD-ROM jukebox.

However, diskfiles on jukeboxes, or more generally, diskfiles on mountable and dismountable file systems may pose problems. Depending on the device driver, dismounting a file system (e.g. MO) may require that all files residing on it are closed. Thus, a mechanism must be supplied to control how the Transbase kernel closes the diskfiles of the database during processing the database.

Therefore, a TBMODE CLOSE ... call is provided to adjust the Transbase kernel with respect to its diskfile closing behaviour, especially for the diskfiles storing BLOBs. By this call, it can be adjusted whether Transbase closes the file

1. after each BLOB fetch
2. after each SQL query

3. after each transaction
4. at exit.

Strategies with higher numbers tend to decrease the cooperativeness and to increase the performance.

Even if files are closed after each query or each BLOB fetch resp., it may happen that more than one files is needed to answer a query or to fetch a BLOB. This may again pose processing problems in the above environments.

Therefore, another call TBMODE PARALLEL OPEN ... call is provided which limits the number of parallel open diskfiles. For example, if the number is restricted to 1, it is assured that a jukebox driver can change the disk if another disk is needed.

It has to be noted that a small limit of parallel open diskfiles may decrease the performance of database processing because of the overhead of diskfile closing and reopening.

7.2.1 TbmodeCloseFileStatement

This statement specifies the behaviour of Transbase with respect to closing the diskfiles of the database.

Syntax:

```
TbmodeCloseFileStatement ::=
    TBMODE CLOSE [ filetype ] FILES AFTER  action

filetype  ::=
    NONBLOB
  | BLOB

action  ::=
    SESSION
  | TRANSACTION
  | QUERY
  | GETBLOB
```

Explanation: If no filetype is specified the specified action applies to all existing diskfiles.

The *action* GETBLOB is only allowed if *filetype* BLOB is specified. If the action act has been specified, Transbase closes the specified diskfiles after an action of type act has been performed.

Before any TBMODE CLOSE ... call has been performed, all files are closed AFTER SESSION or earlier if too many file descriptors are involved.

Example: Specify that Transbase closes all diskfiles after each query:

```
TBMODE CLOSE FILES AFTER QUERY
```

Specify that Transbase closes all dedicated BLOB each time a BLOB has been fetched:

```
TBMODE CLOSE BLOB FILES AFTER GETBLOB
```

7.2.2 TbmodeParallelOpenFileStatement

This statement specifies the maximum number of files that Transbase keeps open in parallel.

Syntax:

```
TbmodeParallelOpenFileStatement ::=
    TBMODE PARALLEL OPEN [ filetype ] FILES number

filetype  ::=
    NONBLOB
  | BLOB

number  ::=
  < natural number greater equal 0>
```

Explanation: If a number `n > 0` is specified Transbase limits the number of parallel open files of type filetype to n. If no filetype is specified, the limit refers separately to open files of both classes.

A specified number 0 means no predefined limit i.e. any limit that has been specified before for that class is not valid any more.

Before any TBMODE PARALLEL OPEN ... call has been performed, no limit is set on the number of open files.

Example: Specify a maximal number of one single open BLOB-dedicated diskfile:

```
TBMODE PARALLEL OPEN BLOB FILES 1
```

Destroy the limit for BLOB-dedicated diskfile:

```
TBMODE PARALLEL OPEN BLOB FILES 0
```

7.3 Tbmode Lockmode Statements

This statement specifies a fixed lock granularity or the default locking strategy of Transbase. These statements *do not* lock objects but influence the lock granularity of the automatic locking of Transbase.

Syntax:

```
LockStatement ::=
    TBMODE LOCKMODE Lockgran

Lockgran ::=
    PAGES  |  TABLES  |  MIXED
```

Explanation: If PAGES is specified, all subsequent locks are on page basis.

If TABLES is specified, all subsequent locks are on table basis. In this mode, at most one lock is set for a table including all its secondary indexes and its BLOBs.

If MIXED is specified, the locking strategy is transferred to Transbase server. This is also the default which is in effect from the begin of the session.

For details of the locking strategy of Transbase see Transbase System Guide.

Note: These statements do not lock any objects. Carefully distinguish the TBMODE LOCKMODE statements from the LOCK and UNLOCK statements which set TABLE locks on random tables.

7.4 Tbmode Plans Statements

This statement enables and disables the generation of evaluation plans.

Syntax:

```
PlanStatement ::=
    TBMODE { PLANS | PROFILES } { ON | OFF }
```

Explanation: If `PLANS` are enabled a textual query execution plan (QEP) is generated when a query is compiled. A second plan is generated after the query is closed. The QEP after query execution contains additional query runtime statistics, such as tuple counters per operator, number of accessed pages per scan etc.

If `PROFILES` are switched on, the QEP after query execution will also include local execution time per operator and total execution time for parts of the plan.

Both plans can retrieved into an application via the appropriate API call.

Switching `PLANS` or `PROFILES` off is equivalent and will cease QEP generation.

Note: Switching query profiling on may cause a significant overhead in the query's total elapsed time, even if the query profiles are not retrieved. The times displayed in the profiles are adjusted to compensate this additional overhead.

Consult the Performance Guide for more details.

Chapter 8

Lock Statements

Transbase locks database objects (i.e. pages or tables or views) automatically. If, however, explicit control of locking is needed, Transbase allows to lock and unlock objects explicitly with table locks.

Two statements, namely a *LockStatement* and an *UnlockStatement*, are provided for that purpose.

Carefully distinguish the LOCK and UNLOCK statements which set TABLE locks on random tables from the TBMODE LOCKMODE statements which influence the lock granularity for automatic locking by Transbase.

8.1 LockStatement

Serves to explicitly lock tables and views.

Syntax:

```
LockStatement ::=
    LOCK  LockSpec [, LockSpec ]...

LockSpec ::=
    Object Mode

Object ::=
    TableName | ViewName

Mode ::=
    READ | UPDATE | EXCLUSIVE
```

Explanation: For each *LockSpec*, the specified lock is set on the specified object. If a view is specified, the lock is effectively set on the underlying base table(s).

For the semantics of locks see Transbase System Guide.

Privileges: The current user needs SELECT-privilege on the specified objects. System tables (the data dictionary) cannot be locked explicitly.

Example:

```
LOCK suppliers READ, quotations UPDATE
```

8.2 UnlockStatement

Serves to remove a READ lock.

Syntax:

```
UnlockStatement ::=
    UNLOCK object

Object ::=
    TableName | ViewName
```

Explanation: The specified object is unlocked, i.e. implicitly set or explicitly requested locks are removed. Error occurs if the object is not locked or if the object is update locked, i.e. an *InsertStatement*, *UpdateStatement*, *DeleteStatement* or an explicit *LockStatement* with UPDATE or EXCLUSIVE mode has been issued within in the transaction.

Chapter 9

The Data Types Datetime and Timespan

9.1 Principles of Datetime

The data type DATETIME is used to describe absolute or periodic points in time with a certain precision. A datetime value is composed of one or more components. For example, the birthday of a person consists of a year, a month and a day and is an absolute point in time with a certain precision. If the hour and minute of the birth is added, then the absolute point in time is described with a higher precision. Examples for periodic points in time are the 24-th of December, the birthday of a person without the year indication, or 12:00:00 (twelve o'clock).

9.1.1 RangeQualifier

The occupied components of a datetime value constitute its range. The components have symbolic names. All names occur in 2 equivalent variants which come from a original Transbase notation and from the evolving SQL2 standard. An overview is given in table 9.1.

The range indices MS, ..., YY are ordered. MS is the smallest, YY is the highest range index. An explicit range qualifier is written as `[ub:lb]` where ub is the upperbound range and lb is the lowerbound range. For example `[YY:DD]` is a valid range qualifier. `[DD:YY]` is an invalid range qualifier.

Upperbound and lowerbound range may be identical, i.e. of the form `[ab:ab]`. A datetime value with such a range has a single component only. The corresponding range qualifier can be abbreviated to the form `[ab]`.

Notation	Meaning	Allowed Values
YY	year	1 - 32767
MO	month	1 - 12
DD	day	1 - 31
HH	hour	0 - 23
MI	minute	0 - 59
SS	second	0 - 59
MS	millisecond	0 - 999

Table 9.1: Ranges of Datetime Components

9.1.2 SQL2 Compatible Subtypes

The SQL2 standard defines the following subtypes which also are supported by Transbase:

SQL2 Type	Alternative Transbase Notation
DATE	DATETIME[YY:DD]
TIME	DATETIME[HH:SS]
TIMESTAMP	DATETIME[YY:MS]

Table 9.2: Additional SQL2 Types for Datetime

For each case, the notations are equivalent.

Note that there are types that can be formulated in Transbase but have no equivalent SQL2 standard formulation. DATETIME[MO:DD] describes yearly periodic datetimes based on day granularity, e.g. birthdays. DATETIME[HH:MI] describes daily periodic datetimes based on minute granularity, e.g. time tables for public traffic.

9.1.3 DatetimeLiteral

DatetimeLiteral defines the syntax for a constant value inside the SQL query text or inside a host variable of type char[]. Transbase supports a variety of literal notations, namely a native Datetime literal representation with maximal flexibility, the SQL2 conformant representation and a notation compatible with the ODBC standard. All 3 variants are sufficiently explained by the following examples. A fourth variant is supported in a Transbase spoolfile.

Example: A datetime value with highest possible precision:

Variant	Notation
Native TB	DATETIME[YY:MS](2002-12-24 17:35:10.25)
Native TB	DATETIME(2002-12-24 17:35:10.25)
SQL2	TIMESTAMP '2002-12-24 17:35:10.025'
ODBC	{ ts '2002-12-24 17:35:10.025' }
Spoolfile only	'2002-12-24 17:35:10.25'

- In the native Transbase notation, the MS component (like all other components) gives the total number of milliseconds. In contrast, in SQL2 and ODBC notation, the dot separator between SS and MS has the meaning of a fractional point.
- All variants except ODBC are also supported in a spoolfile if enclosed in single quotes. Thereby the quotes in the SQL2 variant has to be escaped by a backslash.
- As a general rule, in the native Transbase notation, the range specification [upb:lwb] can be omitted if upb is YY.

Example: A datetime value with day precision:

Variant	Notation
Native TB	DATETIME[YY:DD](2002-12-24)
Native TB	DATETIME(2002-12-24)
SQL2	DATE '2002-12-24'
ODBC	{ d '2002-12-24' }
Spoolfile only	'2002-12-24'

Example: A datetime value with time components only:

Variant	Notation
Native TB	DATETIME[HH:SS](17:35:10)
SQL2	TIME '17:35:10'
ODBC	{ t '17:35:10' }
Spoolfile only	'17:35:10'

There are literals which are only representable in native Transbase notation, because SQL2 (as well as ODBC) only supports subtypes of the most flexible DATETIME type. See the following examples:

Example: The 24-th of December:

```
DATETIME[MO:DD](12-24)
```

Example: Twelve o'clock:

```
DATETIME[HH](12)
```

Note the following rules illustrated by the examples above:

- The range qualifier can be omitted if and only if the upperbound range is YY.
- If upperbound and lowerbound range are identical, the range qualifier can be abbreviated to the form [YX].

9.1.4 Valid Datetime Values

Independent of the notation syntax, some datetime values are not accepted as legal values:

- If MO and DD are inside the range, then the DD value must not exceed the highest existing day of the MO value.
- If YY and DD are inside the range then leap year rules and rules of the julian and gregorian time periods apply.

Example:

```
DATETIME[MO:DD](4-31)
    -- Illegal, no date with such components exists.

DATETIME[MO:DD](2-29)
    -- Legal, there exist dates with such components.

DATETIME(1988-2-29)
    -- Legal : leap year

DATETIME(1900-2-29)
    -- Illegal : no leap year

DATETIME(1582-10-14)
    -- Illegal :
    -- When switching from Julian to Gregorian calendar
    -- this date has been skipped.
```

9.1.5 Creating a Table with Datetimes

As an example, a table is created for personswith w The type specifier for a datetime field of a table in a *CreateTableStatement* consists of the keyword DATETIME followed by a *RangeQualifier*.

Example:

```
CREATE TABLE myfriends
(   name       CHAR(*),
    birthday   DATETIME [YY:DD],  -- alternative syntax: DATE
    firstmet   DATETIME [YY:HH]   -- no alternative
)
```

This table would be appropriate to describe persons with their name and birthday and the time when met first or talked to first.

Note that although all datetime values in the table are exactly of the specified format, it is possible to insert tuples with datetime fields of a different precision. Implicit conversions (CAST operators) then apply as described in the following chapters.

9.1.6 The CURRENTDATE/SYSDATE Operator

An operator CURRENTDATE is provided which delivers the actual date and time. SYSDATE is a synonym for CURRENTDATE. The result type is DATETIME[YY:MS] or TIMESTAMP but note that the effective precision may be less (e.g. `[YY:SS]`) depending on the system clock of the underlying machine. In the latter case, it is tried, however, to set the MS field in such a way that even several successive calls of CURRENTDATE *do not* deliver the same date (see also below).

The operator CURRENTDATE may be used wherever a datetime literal can be used.

CURRENTDATE may also appear as value in a spool file.

When used in a statement the CURRENTDATE operator is evaluated only once and its resulting value remains the same during the whole execution of the statement.

In a TBX or ESQL application program which runs queries with the EVAL call or FETCH statement, resp., the CURRENTDATE operators are evaluated when the first EVAL call or FETCH statement is executed, resp.

As long as a cursor is open, any interleaving cursor sees the same CURRENTDATE result as the already open cursor. In other words, a consistent view of the

CURRENTDATE is provided for interleaving queries inside an application. In all other situations (non interleaving queries), it is tried to evaluate successive calls of CURRENTDATE such that different results are delivered (see above).

9.1.7 Casting Datetimes

A datetime value can be casted to a different range. A cast operation can be performed explicitly by the CAST operator or implicitly occurs before an operation with the datetime value is performed (see also Chapter 2.5 Type Exceptions and Overflow).

In an explicit CAST operation, the syntax of the type specifier is the same as the one used in a *CreateTableStatement*.

Example:

```
CURRENTDATE CAST DATETIME[YY:DD] -- current vear/month/day
CURRENTDATE CAST DATE            -- equivalent to above
CURRENTDATE CAST DATETIME[YY:MO] -- current vear and month
CURRENTDATE CAST DATETIME[HH:SS] -- current hour/minute/second
CURRENTDATE CAST TIME            -- equivalent to above
```

In the sequel, the range boundaries of the value to be casted are called the upperbound source range and the lowerbound source range. The range boundaries of the target range are called the upperbound target range and the lowerbound target range.

The rules to construct the result datetime value from the given one are as follows:

DTC1: All components having a range index smaller than the source lowerbound range are set to the smallest possible value (0 for MS, SS, MI, HH and 1 for DD, MO, YY).

DTC2: All components having a range index higher than the source upperbound range are set to the corresponding components of CURRENTDATE.

DTC3: The other components (i.e. having range index between source lowerbound index and source upperbound index) are set as specified by the source datetime fields.

Example:

```
DATETIME [HH] (12) CAST DATETIME [YY:MS]
```

yields

```
DATETIME [YY:MS] (1989-6-8  12:0:0.0)
```

assuming that the CURRENTDATE is

```
DATETIME [YY:MS] (1989-6-8...)
```

Example:

```
DATETIME [YY:MO] (1989-6) CAST DATETIME [MO:DD]
```

yields

```
DATETIME [MO:DD] (6-1)
```

DD is set to the smallest possible value, namely 1.

Example: The following statement would insert a tuple into the sample table Person:

```
INSERT INTO Person VALUES
( 'Smith', DATETIME (1953-6-8), DATETIME [HH](14) )
```

The correct datetime value for the field "talked" would automatically be generated provided that the statement is run on the same day as the meeting has occurred.

9.1.8 TRUNC Function

The TRUNC function is a shortcut to cast a datetime value to the type DATE, i.e. DATETIME[YY:DD]

Example:

```
TRUNC ( DATETIME [YY:MS] (1989-6-8  12:0:0.0) )
```

yields

```
DATETIME [YY:DD] (1989-6-8)
```

9.1.9 Comparison and Ordering of Datetimes

Datetime values are totally ordered. A datetime d1 is greater than d2 if d1 is later in time than d2.

The SQL operators `<=`, `<`, `=` etc. as used for arithmetic types are also used for comparison of datetimes.

If datetimes values with different ranges are compared, they are implicitly casted to their common range before the comparison is performed. The common range is defined by the maximum of both upperbound and the minimum of both lower-bound ranges. Note, however, special rules for CURRENTDATE as described below!

Thus, it is possible that one or both datetime values are implicitly casted according to the casting rules described in the preceding chapter.

Example: The DATETIME comparison

```
DATETIME[YY:MI](1989-6-8 12:30] = DATETIME[YY:DD](1989-6-8)
```

yields FALSE, because the second operand is implicitly casted to the value `DATETIME[YY:MI] (1989-6-8 00:00)`

Example: The comparison

```
DATETIME[MO:DD](6-8) = DATETIME[YY:DD](2000-6-8)
```

will yield TRUE in the year 2000 and FALSE in other years.

Example: To retrieve all persons who have been met since February this year:

```
SELECT * FROM Persons
WHERE talked >= DATETIME [MO](2)
```

Example: To retrieve all persons whose sign of the zodiac is Gemini:

```
SELECT * FROM Persons
WHERE birthday CAST DATETIME [MO:DD]
BETWEEN DATETIME [MO:DD] (5-21)
AND DATETIME [MO:DD] (6-20)
```

Note that the CAST operator applied to birthday is necessary to restrict the common range to [MO:DD]. If the explicit CAST were omitted, the common range would be [YY:DD] and the constant comparison operators would be extended by the current year so that the query would not hit any person.

Example: To retrieve all persons ordered by their age (i.e. ordered by their birthday descendingly):

```
SELECT * FROM Persons
ORDER BY birthday DESC
```

An exception of the type adaption rule is made for the CURRENTDATE operator. In comparisons (=, <>, <, <=, >, >=) the CURRENTDATE value is automatically adapted to the range of the comparison operand. In most situations this is useful and avoids explicit CAST operations.

Example:

```
SELECT * FROM Persons
WHERE talked CAST DATETIME[YY:DD] = CURRENTDATE
```

This example retrieves all tuples whose field value "talked" matches the current day (year, month, day). Without the type adaption rule for CURRENTDATE, one would also have to cast CURRENTDATE on the range [YY:DD].

9.2 Principles of Timespan and Interval

The data type TIMESPAN (and INTERVAL, as a SQL2 conformant variant) is used to describe distances between absolute or periodic points in time with a certain precision. Examples for TIMESPAN values are the result times of a sports event (measured in hour, minutes, seconds and/or milliseconds), the average life time of a material or product or the age of a person.

9.2.1 Transbase Notation for Type TIMESPAN

The concepts of components, range and range indices are similar to the type DATETIME. The following example shows the strong syntactic analogies between DATETIME and TIMESPAN. However, the semantics are clearly different:

- DATETIME[HH:MI] : suitable for points in time which are periodic on a daily basis (for example taking off time for a flight).
- TIMESPAN[HH:MI] : suitable for describing time intervals on a minute precision for example duration of a flight.

The following important rule applies:

- The range of a TIMESPAN must not span the MO-DD border.

This means that the ranges of all TIMESPAN types must either be inside [YY:MO] or inside [[DD:MS]. For example, the following type definitions are illegal:

- TIMESPAN[YY:DD] – illegal
- TIMESPAN[MO:HH] – illegal

The reason is that the number of days is not the same for all months. So the arithmetic rules for timespan calculations would be compromised.

The set of allowed values on the components are also different from DATETIME. Obviously, the day component of a TIMESPAN value may have the value 0 whereas a DATETIME value containing a day component shows a value ¿= 1. The legal values in a TIMESPAN are shown in table 9.3.

9.2.2 SQL2 Conformant INTERVAL Notation for TIMESPAN

The standard SQL2 notation uses keyword INTERVAL (opposed to TIMESPAN) and different range identifiers with a different syntactic encapsulation. The following examples show the notation in contrast to the TIMESPAN notation.

SQL2 conformant notation	TB native notation
INTERVAL YEAR	TIMESPAN[YY]
INTERVAL YEAR(4) TO MONTH	TIMESPAN[YY:MO]
INTERVAL DAY	TIMESPAN[DD]
INTERVAL DAY(5)	TIMESPAN[DD]
INTERVAL HOUR TO SECOND	TIMESPAN[HH:SS]
INTERVAL HOUR TO SECOND(3)	TIMESPAN[HH:MS]
INTERVAL SECOND(3)	TIMESPAN[SS]
INTERVAL SECOND(5,3)	TIMESPAN[SS:MS]

If SQL2 notation is used, then the type is internally mapped to the corresponding TIMESPAN type. Thereby the optional precision on the start range is ignored.

If the end range is SECOND, then a precision indicates a fractional part so the end range effectiveley becomes milliseconds (MS).

If SECOND is start range (thereby automatically also end range) then a simple precision like (3) is ignored like in all start ranges - especially this precision does not specifiy a fractional part so the mapping is to SS.

If SECOND is start range (thereby automatically also end range) then a specification of a fractional part must be given as (m,n) as it is done in the last example.

9.2.3 Ranges of TIMESPAN Components

On the upperbound range of a TIMESPAN value, always values from 0 through MAXLONG are allowed.

On all components different from the upperbound components only those values are allowed which are below a unity of the next higher component. The allowed values are shown in the table.

By these rules, it is possible for example to express a time distance of 3 days, 1 hour and 5 minutes as 73 hours, 5 minutes i.e. as a TIMESPAN[HH:MI]. However, it is illegal to express it as 72 hours and 65 minutes.

Transbase Notation	SQL2 Notation	Allowed values if not upperbound range
YY	YEAR	0 - MAXLONG
MO	MONTH	0 - 11
DD	DAY	0 - MAXLONG
HH	HOUR	0 - 23
MI	MINUTE	0 - 59
SS	SECOND	0 - 59
MS	–	0 - 999

Table 9.3: Ranges of Timespan Components

9.2.4 TimespanLiteral

There are 2 variants for TIMESPAN literals which correspond to the 2 variants of TIMESPAN type definition (TIMESPAN and INTERVAL). The following table shows the relevant examples in both notations.

SQL2 conformant notation	TB native notation
INTERVAL '2-6' YEAR TO MONTH	TIMESPAN[YY:MO](2-6)
INTERVAL '2:12:35' HOUR TO SECOND	TIMESPAN[HH:SS](2:12:35)
INTERVAL '2 12' DAY TO HOUR	TIMESPAN[DD:HH](2 12)
INTERVAL -'1' YEAR	- TIMESPAN[YY](1)
INTERVAL -'4.5' SECOND	- TIMESPAN[SS:MS](4.500)

Table 9.4: Timespan Literals in Native and SQL2 Notation

Note that negative TIMESPANs are reasonable (e.g. as the result of a subtraction of a DATETIME value from a smaller DATETIME value). In SQL2 syntax, literals with a negative value incorporate the '-' sign within the literal syntax whereas in

Transbase native notation the '-' sign is written (as a separate token) in front of the TIMESPAN token. See also chapter 9.2.5 Sign of Timespans.

9.2.5 Sign of Timespans

A timespan value is positive or zero or negative. It is zero if all components of its range are zero. A negative timespan may result from a computation (see the following chapters) or can also be explicitly represented as a timespan literal prefixed with an unary '-' (in terms of the TB/SQL grammar this is an *Expression*).

Example: The following literal denotes a negative timespan of 3 hours and 29 minutes.

```
- TIMESPAN [HH:MI] (3:29)       -- Transbase native notation
  INTERVAL -'3:29' HOUR TO MINUTE   -- SQL2 conformant syntax
```

Note: It is illegal to attach a '-' sign to any component of a timespan literal.

9.2.6 Creating a Table containing Timespans

The type specifier for a timespan field of a table in a *CreateTableStatement* consists of a TIMESPAN type specifier either in Transbase native syntax or in SQL2 conformant syntax.

Example:

```
CREATE TABLE Marathon
(    name   CHAR(*)
     time   TIMESPAN [HH:MS] -- or INTERVAL HOUR TO SECOND(3)
)
```

Note that although all timespan values in the table are exactly of the specified format, it is possible to insert tuples with timespan fields of a different precision. Implicit conversions (CAST operators) then apply as described in the following chapters.

9.2.7 Casting Timespans

Similarily to datetimes, a timespan value can be explicitly or implicitly casted to a target range. Timespan casting, however, has a complete different semantics than

datetime casting (recall Chapter Datetime Casting). A timespan cast transfers a value into another unit by keeping the order of magnitude of its value unchanged - however a loss of precision or overflow may occur.

The following rules and restrictions apply:

TSC1: The target range must be valid, i.e. it must not span the MO-DD border.

TSC2: The target range must be compatible with the source range, i.e. both ranges must be on the same side of the MO-DD border.

TSC3: If the lowerbound target range is greater than the lowerbound source range then a loss of precision occurs.

TSC4: If the upperbound target range is smaller than the upperbound source range then the component on the upperbound target range is computed as the accumulation of all higher ranged components. This may lead to overflow.

Example: The following literal violates TSC1 and therefore is illegal.

```
TIMESPAN [DD] (90) CAST TIMESPAN [MO:DD]
```

Example: The following literal violates TSC2 and therefore is illegal.

```
TIMESPAN [DD] (90) CAST TIMRSPAN [MO]
```

Example:

```
TIMESPAN [MO] (63) CAST TIMESPAN [YY:MO]
```

yields

```
TIMESPAN [YY:MO] (5-3)
```

Example:

```
TIMESPAN [YY:MO] (5-3) CAST TIMESPAN [MO]
```

yields

```
TIMESPAN [MO] (63)
```

An accumulation (without overflow) occurred.

Example:

```
TIMESPAN [SS] (3666) CAST TIMESPAN [HH:MI]
```

yields

```
TIMESPAN [HH:MI] (1:1)
```

Loss of precision has occurred.

Example:

```
TIMESPAN [DD:MI] (3 12:59) CAST TIMESPAN [HH]
```

yields

```
TIMESPAN [HH] (84)
```

Accumulation (without overflow) and loss of precision have occurred.

Example:

```
TIMESPAN [DD] (365) CAST TIMESPAN [MS]
```

yields overflow by accumulation on MS.

9.2.8 Comparison and Ordering of Timespans

Like DATETIME values, TIMESPAN values are totally ordered and can be compared, sorted etc. If their ranges differ, they are implicitly casted like datetimes (see Chapter 9.1.9 Comparison and Ordering of Datetimes).

Example: The comparison

```
TIMESPAN [MI](69) = TIMESPAN [HH](1)
```

yields FALSE because the operands implicitly are casted to the values `TIMESPAN[HH:MI](1:3)` and `TIMESPAN[HH:MI](1:0)`, resp.

Example: The comparison

```
TIMESPAN [MI] (69) = TIMESPAN [HH:MI] (1:9)
```

yields TRUE.

Example: To retrieve the winner(s) of Marathon:

```
SELECT * FROM Marathon
WHERE time = (SELECT MIN(time) FROM Marathon)
```

9.2.9 Scalar Operations on Timespan

A timespan value can be multiplied by a scalar and divided by a scalar. The result is again a timespan value with the same range as the input timespan value. The scalar can be any arithmetic value but it is casted to type INTEGER before the operation is performed.

Multiplication: The semantics of multiplication is that all components of the timespan are multiplied and the resulting value is normed according to the rules of valid timespans. Overflow occurs if the upperbound range value exceeds MAXLONG.

Example:

```
TIMESPAN [MI:SS] (30:10) * 10
yields
TIMESPAN [MI:SS] (301:40)
```

Example:

```
TIMESPAN [DD:MS] (1 6:50:07.643) * 4
yields
TIMESPAN [DD:MS] (5 3:20:30.572)
```

Example:

```
TIMESPAN [MI:SS] (214748365:10) * 10
```

yields overflow. Cast it to `[HH:SS]` before multiplication to avoid overflow.

Division: The semantics of division is as follows: first the timespan value is casted to its lowerbound range (a virtual cast which never yields overflow!), then the division is performed as an INTEGER division and then the result is casted back to its original range.

Example:

```
TIMESPAN [YY] (1) / 2
```

yields

```
TIMESPAN [YY] (0)
```

Example:

```
TIMESPAN [YY:MO] (1-5) / 2
```

yields

```
TIMESPAN [YY:MO] (0-8)
```

Example:

```
TIMESPAN [DD:MS] (5 3:20:30.572) / 4
```

yields

```
TIMESPAN [DD:MS] (1 6:50:7.643)
```

9.2.10 Addition and Substraction of Timespans

Two timespans with compatible ranges (see Rule TSC2 in Chapter 9.2.7 Casting Timespans) can be added or substracted. The result is a timespan value whose range is the common range of the input values. The common range is again defined by the maximum of both upperbounds and the minimum of both lowerbounds. The input values are casted to their common range before the operation is performed.

Example:

```
TIMESPAN [DD] (1000) + TIMESPAN [DD] (2000)
```

yields

```
TIMESPAN [DD] (3000)
```

Example:

```
TIMESPAN [YY] (1) + TIMESPAN [MO] (25)
```

yields

```
TIMESPAN [YY:MO] (3-1)
```

Example:

```
TIMESPAN [YY] (1) - TIMESPAN [MO] (27)
```

yields

```
-TIMESPAN [YY:MO] (1-3)
```

A negative timespan is the result in this example.

Example: To retrieve the time difference between the winner and the looser of the Marathon as well as the average time:

```
SELECT MAX(time) - MIN(time), AVG(time)
FROM Marathon
```

9.3 Mixed Operations

9.3.1 Datetime + Timespan, Datetime - Timespan

If a timespan is added to or substracted from a datetime, the result is again a datetime. The range of the result is the common range of the two input operands as defined in the preceding chapter.

Example:

```
DATETIME [YY:DD] (1989-6-26) + TIMESPAN [DD] (30)
```

yields

```
DATETIME [YY:DD] (1989-7-26)
```

Example:

```
DATETIME [HH:MI] (12:28) - TIMESPAN [SS] (600)
```

yields

```
DATETIME [HH:SS] (12:18:00)
```

Example:

```
DATETIME [YY:MO] (1989-2) + TIMESPAN [DD:MI] (3 20:10)
```

yields

```
DATETIME [YY:MI] (1989-2-4 20:10)
```

If the upperbound range of the input datetime value is less than YY, then the datetime is always casted to [YY:lb] before the operation is performed (lb is the lowerbound range of the datetime).

Example:

```
DATETIME [MO:DD] (2-28) + TIMESPAN [HH] (24)
```

yields

```
DATETIME [MO:HH] (2-29 0)
```

if run in a leap year; in other years it would yield

```
DATETIME [MO:DD] (3-1 0)
```

Error occurs if the range of the input timespan is on the left side of the MO-DD border and the range of the input datetime is not completely on the left side of the MO-DD border (i.e. something finer than MO is specified). The reason is that the result of the operation could be an invalid datetime.

Example:

```
DATETIME [YY:DD] (1992-2-29) + TIMESPAN [YY] (1)
```

yields an error because something like

```
DATETIME [YY:DD] (1993-2-29)
```

would be invalid.

Example:

```
DATETIME [YY:MO] (1992-2) + TIMESPAN [YY] (1)
```

yields

```
DATETIME [YY:MO] (1993-2)
```

9.3.2 Datetime − Datetime

If two datetimes are subtracted from each other the result is a timespan.

Example:

```
DATETIME [YY:MO] (1989-3) - DATETIME [YY:MO] (1980-4)
```

yields

```
TIMESPAN [YY:MO] (8-11)
```

The result of a datetime subtraction may, of course, also be negative. Except to one case (see below) the range of the result is again the common range of the two input values. If the input ranges are different the two input values are casted to their common range before the operation is performed.

Example:

```
DATETIME [HH:MI] (12:35) - DATETIME [HH] (14)
```

yields

```
TIMESPAN [HH:MI] (1:25)
```

One slight complication arises if the range of the resulting timespan would span the MO-DD border and thus would be invalid. In this case, the upperbound of the result range is always DD.

Example:

```
DATETIME [YY:DD] (1989-6-26) - DATETIME [YY:DD] (1953-6-8)
```

yields

```
TIMESPAN [DD] (13167)
```

9.4 The WEEKDAY Operator

The WEEKDAY operator is applicable to a datetime whose range contains at least YY, MO and DD. It delivers the corresponding weekday as an INTEGER value between 0 and 6 where 0 means Sunday, 1 means Monday etc. The syntax is the keyword WEEKDAY followed by the keyword OF followed by a datetime.

Example:

```
WEEKDAY OF DATETIME (2000-1-1)
```

yields 6 (meaning Saturday).

Example: To retrieve all persons who have been met on a Sunday

```
SELECT * FROM Persons
WHERE WEEKDAY OF talked = 0
```

9.5 Selector Operators on Datetimes and Timespans

A selector operator extracts a single component from a datetime or timespan value converted into the type INTEGER. A selector operation consists of one of the keywords YY, MO, DD, HH, MI, SS, MS, followed by an expression of type DATETIME or TIMESPAN.

Error occurs if the selector is not inside the range of the value.

Note that selecting a component semantically is not the same as casting to the range of the selector as shown by the examples:

Example:

```
MS OF TIMESPAN [SS:MS] (2.032)
```

yields 32.

Example:

```
TIMESPAN [SS:MS] (2.032) CAST TIMESPAN [MS]
```

yields

```
TIMESPAN [MS] (2032)
```

Note that the selector operator simply extracts one component without regarding the semantics of DATETIME or TIMESPAN. However, the selector operators (as well as the CONSTRUCT operator, see below) are useful because they provide for a bridge between DATETIME/TIMESPAN and the basic arithmetic types of SQL. For example an application program can retrieve the components of datetime or timespan values into integer program variables for further arithmetic processing.

9.6 Constructor Operator for Datetimes and Timespans

The constructor operator (CONSTRUCT) is inverse to the selector operators. It constructs a datetime or timespan value from a list of arithmetic expressions. Syntactically it consists of the keyword CONSTRUCT followed by a syntax which is similar to that of a datetime or timespan literal. However, the components can be arithmetic expressions and are separated by commata. The arithmetic expressions are automatically casted to INTEGER before the CONSTRUCT operator is performed. Let e1, e2, e3 be arithmetic expressions in the following examples.

Example:

```
CONSTRUCT DATETIME [YY:DD] (1986,10,6)
```

is equivalent to

```
DATETIME [YY:DD] (1986-10-6)
```

Example:

```
CONSTRUCT DATETIME [MO:HH] (e1,e2,e3)
```

Example: If omitted the range is assumed to start with YY. The following literal therefore denotes a range of [YY:MO].

```
CONSTRUCT TIMESPAN (e1,e2)
```

Note that if all values are constants, the CONSTRUCT operator is in no way superior to an equivalent datetime or literal representation which is also better readable.

CONSTRUCT is appropriate to build datetime and timespan values from components which are evaluated at runtime. For example, it is very useful for application programs which insert tuples with datetime or timespan values built up at runtime.

Example: In an ESQL program, the following code would be suitable to insert a tuple into the Persons table:

```
EXEC SQL BEGIN DECLARE SECTION;
   char name [30];
   int year, month, day;
EXEC SQL END DECLARE SECTION;

year = 1990;
month = 10;
day = 3;

EXEC SQL INSERT INTO Persons(name,birthday)
VALUES
(:name, CONSTRUCT DATETIME(:year, :month, :day) );
```

The constructor and selector operators together allow to perform every manipulation on datetime and timespan values and also to override the built-in semantics. This may be necessary only occasionally as shown below.

Example: Assume that in the table Persons several values for birthdays have been inserted (falsely) without the century of the year (e.g. 53-6-8 instead of 1953-6-8). The following statement would correct all such entries:

```
UPDATE Persons
SET birthday = CONSTRUCT DATETIME
    (YY OF birthday + 1900,
     MO OF birthday,
     DD OF birthday)
WHERE YY OF birthday < 100
```

In effect, the above statement does not express a semantically reasonable operation on datetimes but a correction of wrong datetime values. Note that this correction cannot be performed by adding a timespan value TIMESPAN [YY] (1900) because of the subtle semantics of the addition of timespans to datetimes.

Chapter 10

The TB/SQL Datatypes BITS(p) and BITS(*)

This chapter describes the TB SQL datatype BITS(p) and BITS(*), which represent bits vectors with fixed or variable size.

10.1 Purpose of Bits Vectors

Bits vectors are suited to represent certain 1-to-many relationships in a very compact manner. Assume a table TI with a field FK of arbitrary type and field FI of type INTEGER or SMALLINT or TINYINT.

User table TI:

FK	FI(INTEGER)
a	1
a	4
a	7
a	8
b	3
b	10
b	11

A representation using bitvectors yields the following table TB with fields FK and FB where FB is of type BITS(*):

Table TB:

FK	FB (BITS(*))
a	0b10010011
b	0b00100000011

The used notation here is that of bits literals (0-1 sequence starting with 0b).

10.2 Creation of Tables with type BITS

The notation in a DDL Statement is analogous to that of CHAR.

Example: Creation of a table TI with a variable sized BITS field:

```
CREATE TABLE TI
(
    FK INTEGER,
    FB BITS(*),
    ...
)
```

Example: Creation of a table with a fixed sized BITS field:

```
CREATE TABLE relb
(
    k INTEGER,
    b BITS(512),
    ...
)
```

The number p in BITS(p) is the number of bits that a value or a field of that type may hold. The maximum number of p is MAXSTRINGSIZE*8-4 , where MAXSTRINGSIZE depends on the pagesize. A value of type BITS(p) or BITS(*) semantically is a series of 0 or 1-bits. The bit positions are numbered and the leftmost position has the number 1.

10.3 Compatibility of BITS, CHAR and BINCHAR

The types CHAR, BINCHAR, BITS are compatible among each other. They form a hierarchy CHAR, BINCHAR, BITS in increasing order (i.e. BITS is the highest of the 3 types).

Analogously to the arithmetic types, the value of the lower level type is automatically casted to the higher level type when an operation requires a higher level type input or when two values of different types are compared or combined.

10.4 BITS and BINCHAR Literals

A BITS literal is a sequence of the digits 0 and 1 prefixed by 0b.

Example:

```
0b0101       -- Type is BITS(4)
0b111100001  -- Type is BITS(9)
```

Inside the 0-1-sequence a positive repetition factor can be used as a shorthand notation for a series of equal bits:

Example:

```
0b0(4)1(5)0
```

is a shorthand notation for

```
0b0000111110
```

A repetition factor is a IntegerLiteral in round brackets.

No computed expression is allowed here. With a little bit care, also BINCHAR literals can be used for constants of type BITS, because BINCHAR is implicitly casted to BITS where needed. Note however that the values are not identical, e.g. the SIZE operator delivers different results.

Example:

```
0xaf08             -- is a BINCHAR literal
0b1010111100001000 -- is a BITS literal
```

They are not identical because

```
SIZE OF 0xaf08             -- delivers 2
SIZE OF 0b1010111100001000 -- delivers 16
```

The following expression, however, is identical to the above BITS literal.

```
0xaf08 CAST BITS(*)
```

A further shorthand notation is given by a dynamic bits constructor MAKEBIT (see below).

Note: When a BINCHAR value (e.g. a Literal) of type BINCHAR(p) is used as input for an operation which requires the type BITS, it is automatically casted to the type BITS(p*8).

10.5 Spool Format for BINCHAR and BITS

The spool format as produced by Transbase is the BincharLiteral representation. The accepted format for spooling from file to tables is BITS Literal as well as BINCHAR Literal.

10.6 Operations for Type BITS

In the following paragraphs, the notations bexpr and iexpr are used. bexpr denotes a value of type BITS(p) or BITS(*). iexpr denotes a value of type TINYINT/SMALLINT/INTEGER. Both notations stand for constants as well as for computed expressions, e.g. subqueries.

10.6.1 Bitcomplement Operator BITNOT

Syntax:

```
BITNOT bexpr
```

Explanation: Computes the bitwise complement of its operand. The result type is the same as the input type.

Example:

```
BITNOT 0b001101           -- yields 0b110010
```

10.6.2 Binary Operators BITAND , BITOR

Syntax:

```
bexpr1 BITAND bexpr2
bexpr1 BITOR  bexpr2
```

Explanation: BITAND computes the bitwise AND, BITOR the bitwise OR of its operands. The shorter of the two input operands is implicitly filled with 0-bits up to the length of the longer input operands. If one of bexpri is type BITS(*) then the result type is also BITS(*) else the result type is BITS(p) where p is the maximum of the input type lengths.

Example:

```
0b1100 BITAND 0b0101       -- yields 0b0100
0b1100 BITOR  0b0101       -- yields 0b1101
```

10.6.3 Comparison Operators

All comparison operators (`< , <= , = , <> , > , >=`) as known for the other Transbase types are also defined for BITS. Length adaption is done as for BITAND and BITOR. A BITS value b1 is greater than a BITS value b2 if the first differring bit is 1 in b1 and 0 in b0.

10.6.4 Dynamic Construction of BITS with MAKEBIT

Syntax:

```
MAKEBIT ( iexpr1, [ , iexpr2 ] )
```

Explanation: If both iexpr1 and iexpr2 are specified: iexpr1 and iexpr2 describe a range of bit positions. Both expressions must deliver exactly one value which is a valid bit position (¿= 1). MAKEBIT constructs a bits value which has 0-bits from position 1 to iexpr1-1 and has 1-bits from position iexpr1 to iexpr2.

If only iexpr1 is specified: iexpr1 describes one bit position or (in case of a subquery) a set of bit positions. MAKEBIT constructs a bits value which has 1-bits exactly on those positions described by iexpr1.

The result type is BITS(*).

Example:

```
MAKEBIT ( 3 , 7 )          -- yields 0b0011111
MAKEBIT ( SELECT ... )     -- yields 0b00101001
                           -- assuming the subquery delivers
                           -- values 3, 5, 8
```

10.6.5 Counting Bits with COUNTBIT

Syntax:

```
COUNTBIT ( bexpr )
```

Explanation: Returns number of 1-bits in bexpr, i.e. a number ≥ 0. The result type is INTEGER.

Example:

```
COUNTBIT ( 0b01011 )          -- yields 3
```

10.6.6 Searching Bits with FINDBIT

Syntax:

```
FINDBIT ( bexpr [ , iexpr ] )
```

Explanation: If iexpr is not specified it is equivalent to 1. If iexpr is greater or equal to 1, FINDBIT returns the position of the iexpr-th 1-bit in bexpr if it exists else 0. If iexpr is 0, FINDBIT returns the position of the last 1-bit in bexpr if there exists one else 0.

The result type is INTEGER.

Example:

```
FINDBIT ( 0b001011 , 1 )    -- yields 3
FINDBIT ( 0b001011 , 2 )    -- yields 5
FINDBIT ( 0b001011 , 4 )    -- yields 0
FINDBIT ( 0b001011 , 0 )    -- yields 6
```

10.6.7 Subranges and Single Bits with SUBRANGE

Syntax:

```
bexpr SUBRANGE ( iexpr1 [ , iexpr2 ] )
```

Explanation: If iexpr2 is specified then SUBRANGE constructs from bexpr a bits value which consists of the bits from position iexpr1 until position iexpr2 (inclusive). If iexpr2 exceeds the highest bit position of bexpr then 0-bits are implicitly taken.

If iexpr2 is not specified then SUBRANGE returns the iexpr1-th bit from bexpr as a INTEGER value (0 or 1).

In all cases iexpri must be valid bit positions ($\geq$1).

The result type is BITS(*) if iexpr2 is specified else INTEGER.

Example:

```
0b00111011 SUBRANGE (4, 6)    --> 0b110     (BITS(*))
0b00111011 SUBRANGE (6, 10)   --> 0b01100   (BITS(*))
0b00111011 SUBRANGE (2)       --> 0         (INTEGER)
0b00111011 SUBRANGE (3)       --> 1         (INTEGER)
```

10.7 Transformation between Bits and Integer Sets

Two operations are defined which serve to transform 1-n relationships into a compact bits representation and vice versa. Assume again the sample tables TI and TB given in chapter 10.1 Purpose of Bits Vectors. The following picture illustrates how the tables can be transformed into each other by an extension of the GROUP BY operator and a complementary UNGROUP BY operator. The operators are explained in detail in the following sections.

FK	FI(INTEGER)
a	1
a	4
a	7
a	8
b	3
b	10
b	11

FK	FB (BITS(*))
a	0b10010011
b	0b00100000011

10.7.1 Compression into Bits with the SUM function

The set function SUM, originally defined for arithmetic values, is extended for the type BITS(p) and BITS(*). For arithmetic values, SUM calculates the arithmetic sum over all input values. Applied to BITS values, SUM yields the BITOR value over all input values where a start value of 0b0 is assumed.

In combination with a GROUP BY operator and MAKEBIT operator, the table TI can be transformed to the table TB (see Chapter 10.1 Purpose of Bits Vectors):

```
SELECT FK , SUM ( MAKEBIT ( FI ) )
FROM RI
GROUP BY FK
```

Also the notation OR instead of SUM is legal here.

10.7.2 Expanding BITS into Tuple Sets with UNGROUP

Given a table of the shape of TB (i.e. with at least one field of type BITS(p) or BITS(*), one can expand each tuple into a set of tuples where the BITS field is replaced by an INTEGER field.

An UNGROUP operator is defined which can be applied to a field of type BITS(p) or BITS(*).

The following statement constructs table TI from table TB (see chapter 10.1 Purpose of Bits Vectors):

```
SELECT * FROM RB
UNGROUP BY FB
```

The UNGROUP BY operator can be applied to exactly one field and this field must be of type BITS.

For completeness, the full syntax of a SelectExpression (QueryBlock) is:

(6) `SelectClause`

(1) `FromClause`

(2) `[ WhereClause ]`

(3) `[ UngroupClause ]`

(4) `[ GroupClause ]`

(5) `[ HavingClause ]`

```
UngroupClause ::=
    UNGROUP BY FieldReference
```

The numbers at the left margin show the order in which the clauses are applied. It shows that the *UngroupClause* takes the result of the *WhereClause* as input: it constructs from each input tuple t a set of tuples where the BITS value of t at position *FieldName* is replaced by INTEGER values representing those bit positions of t which are set to 1.

Note: It will rarely occur that the *UngroupClause* as well as the *GroupClause* are specified.

Chapter 11

The Data Type BLOB (Binary Large Object)

This chapter gives an overview of BLOBs in Transbase. Described are the DDL and SQL kernel language extensions. Note that the facilities on the programming interfaces TBX and TB/ESQL are described in detail in the corresponding manuals.

11.1 Inherent Properties of BLOBs

BLOB is a data type for fields of tables. Arbitrary many fields of a table can be declared with type BLOB. BLOBs are variable sized.

11.1.1 Overview of operations

Transbase does not interpret the contents of a BLOB. Each field of type BLOB either contains the NULL value or a BLOB object. The only operations on BLOBs are creation, insertion, update of a BLOB, testing a BLOB on being the NULL value, extracting a BLOB via the field name in the SELECT clause, extracting a subrange of a BLOB (i.e. an adjacent byte range of a BLOB), and extracting the size of a BLOB.

11.1.2 Size of BLOBs

BLOB fields are variable sized. The size of a BLOB object is restricted to the positive byte range of a 4-byte integer (2 power 31 Bytes) minus some per-page-overhead of about 1%. The sum of sizes of all BLOBs of one relation is restricted to 2 power 42 Bytes (about 4 Terabytes) minus some overhead of about 1.5

11.2 BLOBs and the Data Definition Language

The keyword BLOB describes the data type of a BLOB field in the CreateTableStatement.

Example:

```
CREATE TABLE GRAPHIK
(
    GRAPHIK_NAME  CHAR(20),
    GRAPHIK_TYP   INTEGER,
    IMAGE         BLOB
)
KEY IS GRAPHIK_NAME
```

A BLOB field can be declared NOT NULL. No secondary index can be built on a BLOB field.

11.3 BLOBs and the Data Manipulation Language

11.3.1 BLOBs in SELECT Queries

A SELECT Query that contains result fields of type BLOB prepares the database kernel to deliver the BLOB objects, however, it depends on the environment how the BLOB objects are handled; TB/ESQL e.g. automatically fetches the BLOB objects into file or main memory, TBX requires an extra call to fetch the BLOBs, with TBI the BLOBs cannot be fetched in the current implementation.

BLOB fields can appear in the ExprList of the SELECT clause of a QueryBlock, either explicitly or via the '*' notation.

No operators (except the subrange operator and the SIZE OF operator, see below) are allowed on BLOB fields.

Example:

```
SELECT GRAPHIK_NAME, IMAGE
FROM GRAPHIK
```

With the SUBRANGE operator (n,m) where n and m are positive integers, a part of a BLOB can be retrieved. The following example retrieves the first 100 bytes of all image fields:

Example:

```
SELECT GRAPHIK_NAME, IMAGE SUBRANGE (1,100)
FROM GRAPHIK
```

With the SIZE OF operator, one can retrieve the size in bytes of a BLOB field. SIZE OF delivers NULL if the field is NULL. The following example retrieves the sizes of all BLOB objects in the sample table.

Example:

```
SELECT GRAPHIK_NAME, SIZE OF IMAGE
FROM GRAPHIK
WHERE IMAGE IS NOT NULL
```

A BLOB field can appear in the SearchCondition of the WHERE clause only inside a NullPredicate. It is important to note that the DISTINCT clause in the ExprList of a SELECT clause does not eliminate 'identical' BLOB objects. This means that any two BLOB objects are considered different in the database even if their contents actually are identical. Analogously, the GROUP BY operator if applied BLOB objects forms one GROUP for every BLOB object.

BLOB objects have no meaningful order for the user. It is not an error to apply the ORDER BY clause to BLOB fields but the ordering refers to internal BLOB addresses and thus the result is of no use in the user's view.

11.3.2 BLOBs in INSERT Queries

With the interactive interfaces UFI and TBI it is not possible to insert new BLOB objects into relations (but it is possible to use SPOOL commands, see below). However, the programming interfaces TBX and TB/ESQL offer mechanisms to insert BLOB objects (from file or from main memory). These mechanisms are described in the corresponding manuals.

11.3.3 Spooling BLOBs

The SPOOLing of tables with BLOB objects is described in Chapter 'Spooling Blob Objects' within the Section 'The Data Spooler'.

Chapter 12

Fulltext Indexes

Transbase fulltext search is supported on fields of type BLOB, CHAR(p) and CHAR(*).

12.1 FulltextIndexStatement

A *FulltextIndexStatement* is provided which creates a fulltext index on one field.

Syntax:

```
FulltextIndexStatement ::=
    CREATE [POSITIONAL] FULLTEXT INDEX IndexName
    [FulltextSpec] ON TableName (FieldName)
    [ScratchArea]

FulltextSpec ::=
    [ { Wordlist | Stopword } ]  [ Charmap]  [ Delimiters ]

Wordlist ::=
    WORDLIST FROM WTableName

Stopword ::=
    STOPWORDS FROM STableName

Charmap ::=
    CHARMAP FROM CTableName
```

```
Delimiters ::=
    DELIMITERS FROM  DTableName
  | DELIMITERS NONALPHANUM

WTableName, STableName, CTableName, DTableName ::=
    Identifier

ScratchArea ::=
    SCRATCH IntegerLiteral MB
```

Explanation: A fulltext index is the prerequisite for fulltext search on specified field (Fulltext-Predicate). Depending on whether POSITIONAL is specified or not, the fulltext index is called positional index or word index.

A word index allows so called word search whereas a positional index additionally offers so called phrase search. Word search and phrase search are explained below.

Beside the two variants called word index and positional index, fulltext indexes come in three further independent variants. The specifications WORDLIST, STOPWORDS, CHARMAP and DELIMITERS influence the contents of the fulltext index. They are explained below. All four specifications include a *TableName* which is a user supplied table. The contents of the table(s) supply information to the *FulltextIndexStatement* at the time it is performed.

After the statement's execution, the contents of the tables are integrated into the index and the tables themselves do not further influence the behaviour of the created index. They can be dropped by the user if they are not needed any more for other purposes.

The SCRATCH Clause is explained in Chapter 'Scratch Area for Index Creation'.

WORDLIST and STOPWORDS By default, if neither WORDLIST nor STOPWORDS is specified, *all* words from the indexed field are indexed.

By WORDLIST, a positive list of words can be specified, i.e. specified words are indexed *only*.

By STOPWORDS, a negative list of words is specified, i.e. all words *except* those in the stopword list are indexed.

WTable (for WORDLIST) or *STable* (for STOPWORDS) must have one single field of type CHAR(*) or CHAR(p).

The WORDLIST and STOPWORDS variant mutually exclude each other.

If WORDLIST or STOPWORDS are specified, the fulltext index typically becomes much smaller because less words are indexed. On the other hand, if the fulltext predicate contains words which are not indexed, tuples which contain not-indexed words do not appear in the result set.

CHARMAP By specifying CHARMAP, a character mapping algorithm can be supplied. It is specified by first inserting binary tuples into a binary table (let's say *CTable*) with fields CHAR(1) and CHAR(*) and by specifying *CTable* in the CHARMAP clause. For example, the table could contain a mapping from the German 'umlauts' ä, into ae, ö into oe, etc. such that the search need not rely on German keyboards.

'ä'	'ae'
'ö'	'oe'
'ü'	'ue'

Note: The character mapping is automatically supplied on the indexed words as well as on all search arguments in the *FulltextPredicate*. In the example above, the word 'lösen' would be stored as 'loesen' and a search pattern 'lö%' in a query would be transformed to 'loe%'.

It is also possible to specify the empty string as the target string for a certain character. Consequently, this causes all occurrences of that character to be ignored. For example, a tuple in *CTable* of the form

'.'	''

causes all occurrences of dot to be ignored. Thus, the word 'A.B.C.D' would be stored as 'ABCD' (and search for 'A.B.C.D' would hit as well as a search for 'ABCD'). Note, however, that in this example, a missing blank (delimiter, to be exact) after the concluding dot of a sentence would have the undesired effect to combine 2 words into one.

By default, a fulltext index works in case sensitive mode. Case insensitive search can be achieved by supplying a character mapping table which maps each upper case letter to ts corresponding lower case letter.

DELIMITERS The DELIMITERS clause specifies the word processing in the indexing process. If no DELIMITERS clause is specified, the indexing procedure handles each longest sequence of non-white-space characters as one word, i.e. by default, words are separated by white-space characters (blank, tabulator and newline). Also non-printable characters are treated as delimiters.

For example, the preceeding sentence would produce, among others, the words 'specified,' and 'non-white-space'. It is often convenient to supply additional word delimiters like '(' , '.' etc.

Different delimiters can be specified by the DELIMITERS clause. If a Delimiters Table is specified, it must have 1 field of type CHAR(1) or CHAR(*) and must

contain characters (strings of length 1). However, *non-printable* character are always treated as delimiters.

The NONALPHANUM specification provides a shorthand notation for the convenient case that all characters which are not alphanumeric (not 0-9, a-z, A-Z) are to be treated as delimiters.

Note that search patterns in Fulltext Predicates are not transformed with respect to delimiters (in contrast to CHARMAP!).

For example, if default delimiters have been used (white space) and a fulltext predicate contains a search component with a blank (e.g 'database systems'), then no tuple fulfills the predicate. In this case, one would have to formulate a fulltext phrase with two successive words — this is described later.

Example: In all following examples for *CreateIndexStatement*s, let f be a table which contains a BLOB field fb, and wl, sw, del be unary tables containing a wordlist, a stopword list, a delimiter character list, resp. Let cm be a binary table containing a character mapping.

```
CREATE FULLTEXT INDEX fbx
ON f(fb)

CREATE POSITIONAL FULLTEXT INDEX fbx
ON f(fb)

CREATE FULLTEXT INDEX fbx
WORDLIST FROM wl
ON f(fb)

CREATE FULLTEXT INDEX fbx
STOPWORDS FROM sw
CHARMAP FROM cm
DELIMITERS FROM del
ON f(fb)

CREATE FULLTEXT INDEX fbx
DELIMITERS NONALPHANUM
ON f(fb)
```

12.2 Implicit Tables of a Fulltext Index

Each fulltext index has a wordlist which contains the words which have been indexed so far (or, in the case of a WORDLIST clause have been defined as the

positive wordlist). The wordlist can be accessed by SQL statements as a pseudo table via a pseudo name described below.

For each of the STOPWORDS, CHARMAP and DELIMITERS clause, another pseudo table is created and is accessible like a normal table via a pseudo name. These tables should not be confused with the tables *STableName*, *CTableName*, *DTableName* supplied for the *CreateIndexStatement*. The latter are user defined tables which however define the contents of the pseudo tables at statement execution time. Any successive update to these user tables does not have any influence to the index and its pseudo tables.

The names of the pseudo tables are derived from the name of the fulltext index. The table and field names as well as their types are given as follows (assume that the fulltext index has the name fbx):

```
FULLTEXT WORDLIST OF fbx ( word CHAR(*), wno INTEGER )

FULLTEXT STOPWORDS OF fbx ( word CHAR(*) )

FULLTEXT CHARMAP OF fbx ( source CHAR(1), target CHAR(*) )

FULLTEXT DELIMITERS OF fbx ( delimword CHAR(1) )
```

For example, to see the words indexed up so far or to see the valid delimiters (if a DELIMITERS clause had been specified) one could say:

```
SELECT word FROM FULLTEXT WORDLIST OF fbx

SELECT * FROM FULLTEXT DELIMITERS OF fbx
```

The pseudo tables are not recorded in the catalog table 'systable'.

It is also possible to update the internal WORDLIST OF table or STOPWORDS OF table in a restricted manner:

- The allowed update operating on a WORDLIST OF table is DELETE.
- The allowed update operating on a STOPWORDS OF table is INSERT.

By modifications of these internal tables one can influence the indexing behaviour of the fulltext index for future INSERTs into the base table. The current contents of the fulltext index are not changed.

12.3 FulltextPredicate

Search expressions on fulltext-indexed fields are expressed with a FulltextPredicate.

Syntax:

```
FulltextPredicate ::=
    FieldName CONTAINS ( FulltextTerm )

FulltextTerm ::=
    FulltextFactor [ OR FulltextFactor ] ...

FulltextFactor ::=
    FulltextPhrase  [ Andnot FulltextPhrase ] ...

Andnot::=
    AND  |  NOT

FulltextPhrase ::=
    ( FulltextTerm )
  | Atom  [ [ DistSpec ]  Atom ] ...

Atom ::=
    SingleValueAtom
  | MultiValueAtom

SingleValueAtom ::=
    CharLiteral
  | Parameter
  | FtExpression

MultiValueAtom ::=
    ANY ( TableExpression )

DistSpec ::=
    Leftbracket  [ MinBetween , ] MaxBetween  Rightbracket

Leftbracket ::=  [
```

```
Rightbracket ::=  ]


MinBetween , MaxBetween::=
    <Expression of type Integer>

Parameter ::=
    <see Primary>

FtExpression ::=
    <Expression without FieldReference to same block>

CharLiteral ::=
    <literal of type character>
```

Explanation: The *FieldName* of a *FulltextPredicate* must refer to a field which has a fulltext index. The result type of *SingleValueAtom* must be CHAR(n) or CHAR(*).

A *FulltextPredicate* consists of a *FieldName*, the operator CONTAINS and a *FulltextTerm* in parentheses. The *FulltextTerm* is an expression consisting of *FulltextPhrase*s and the operators AND, OR, NOT. The precedence is NOT before AND before OR. Parentheses may be used.

*FulltextPhrase*s are of different complexities. The simplest form is a single *Atom* (e.g. a CHAR literal like 'database' or an application host variable). More complex forms have sequences of *Atom*s separated by *DistSpec*s.

A *FulltextPredicate* whose *FulltextPhrases* all consists of single *Atom*s only, is called a 'word search'.

A *FulltextPredicate* which contains a *FulltextPhrase* which is not a single *Atom* (i.e. contains at least 1 *DistSpec*) is called a 'phrase search'.

Note: If the *FulltextPredicate* is a phrase search then the fulltext index must be a POSITIONAL fulltext index.

A POSITIONAL fulltext index uses about three times the space of a non-positional fulltext index.

Example: The following statements show word searches:

```
SELECT * FROM f WHERE fb CONTAINS ( 'database' )

SELECT * FROM f WHERE fb CONTAINS ( :hvar)        -- ESQL

SELECT * FROM f WHERE fb CONTAINS ( 'data%' AND 'systems' )

SELECT * FROM f WHERE
fb CONTAINS ( 'database' NOT 'object' OR 'SQL' NOT '4GL')
```

Example: The following statements show phrase searches:

```
SELECT * FROM f WHERE
fb CONTAINS ( 'database'   'systems' )

SELECT * FROM f WHERE
fb CONTAINS ( 'object%' [0,1] 'database' 'systems'
OR 'distributed' [1] 'systems' )
```

Wildcards: Wildcard characters '%' and '_' have the same semantics as in the second operand of the LIKE predicate.

Escaping Wildcards: The character '\' is reserved to escape a wildcard. If '\' is needed as a character of a word it must also be escaped. These rules are the same as in the LIKE predicate with a specified ESCAPE '\'.

Word Set of an Atom: An Atom A in a *FulltextPredicate* specifies a word set WS(A) defined as follows.

If Atom A is a SingleValueAtom with result value SV: If the result value of SV does not contain a wildcard then WS(A) consists of SV only, otherwise – SV contains wildcard(s) WS(A) consists of all words matching the pattern SV where matching is defined like in the explanation of the SQL LIKE predicate (with the '\' character as ESCAPE character).

If Atom A is a MultiValueAtom with result set MV: WS(A) is the union of all WS(A') where A' are Atoms for the single elements of MV.

Semantics of Fulltext Predicates:

fb CONTAINS (Atom) yields TRUE if and only if the field fb contains one of the words of WS(Atom), the word set specified by Atom.

fb CONTAINS (Atom1 [n,m] Atom2) where n and m are integer values, then the predicate yields TRUE if and only if the field fb contains a word w1 of the WS(Atom1) and a word w2 of WS(Atom2) and the number of words between w1 and w2 is at least n at at most m.

Atom [m] Atom is equivalent to: Atom [0,m] Atom

A missing distance specification is equivalent to [0]. Especially, a phrase for a series of adjacent words can be simply expressed as
Atom Atom Atom ....

FulltextPhrase1 NOT FulltextPhrase2 delivers TRUE if and only if

fb CONTAINS(FulltextPhrase1) delivers TRUE and

fb CONTAINS(FulltextPhrase2) does not deliver TRUE.

FulltextPhrase1 AND FulltextPhrase2 is equivalent to:

fb CONTAINS(FulltextPhrase1) AND fb CONTAINS(FulltextPhrase2)

FulltextFactor1 OR FulltextFactor2 is equivalent to:

fb CONTAINS(FulltextFactor1) OR fb CONTAINS(FulltextFactor2)

Note: Do not omit the separating blank characters in the series of words of a phrase search! For example, consider the following specification:

```
fb CONTAINS( 'object''database''systems' )
```

effectively searches for one word consisting of 23 characters including two single apostrophes. Note that the rules for SQL string literals apply.

12.4 Examples and Restrictions

In the following examples let F be a table with field fb of type BLOB where a fulltext index on fb has been created. Let WT be a table with a field word of type CHAR(*).

12.4.1 Examples for Fulltext Predicates

(1)
```
SELECT * FROM F
WHERE fb CONTAINS ( 'database'  [ 0,1 ]  'systems' )
-- delivers tuples where fb contains
-- the series of the two specified words
-- with at most one word in between.
```

(2)
```
SELECT * FROM F
WHERE fb CONTAINS ( 'object'   'database'  'systems' )
-- yields TRUE for tuples where "fb"
-- contains the series of the three specified words.
```

(3)
```
SELECT word FROM FULLTEXT WORDLIST WT
WHERE EXISTS
(SELECT * FROM F WHERE fb CONTAINS ( WT.word ) )
-- delivers the values of "word"
-- which occur as words in the field "fb" of any tuple of F.
```

(4)
```
SELECT * FROM F
WHERE fb contains ( ANY ( SELECT  LOWER(word)  FROM WT) )
-- delivers the tuples of "F"
-- whose "fb" value contains at least one lowercase word
-- of the word set described by field "word" of table "WT".
```

(3) shows an application of a *SingleValueAtom* where the *Atom* is not a simple *Literal* or *Primary*.

(4) shows an application of a *MultiValueAtom*.

12.4.2 Restrictions for Fulltext Predicates

Although the fulltext facility of Transbase is of considerable power, it also exhibits some syntactic restrictions which, however, can be circumvented.

Restriction 1: A *SingleValueAtom* must not start with a '('.

For example, a *SingleValueAtom* of the form

```
( 'a' ||  :hvar) CAST CHAR(30)
```

is illegal because the '(' syntactically introduces a *FulltextTerm* of *FulltextPhrase*.

In these very rare cases replace the *SingleValueAtom* SVA by

```
'' || (SVA)
```

which is a string concatenation with the empty string.

Restriction 2: An *Atom* must not contain a field reference to the same block where the fulltext table occurs.

Assume a table FT with fulltext field fb and fields fk and fc, where fc is of type CHAR(*) and fk is the key.

The following is illegal:

```
SELECT * FROM FT
WHERE fb CONTAINS (fc) -- ILLEGAL
```

This query must be formulated by using a subquery which combines FT with FT via the key fk:

```
SELECT * FROM FT ftout
WHERE EXISTS (
   SELECT  *
   FROM FT ftin
   WHERE ftout.fk = ftin.fk
   AND ftin.fb CONTAINS (ftout.fc)
   )
```

This query is legal because an outer reference in a fulltext atom is legal.

12.5 Performance Considerations

12.5.1 Search Performance

The fulltext index enables very good search times for fulltext searches. It, however, also causes some Performance limitations in database processing. This is described in the following chapters.

12.5.2 Scratch Area for Index Creation

Creation of a Fulltext Index is a time-consuming task if the base table and/or the field values (BLOBs) are large.

The processing time considerably depends on the amount of available temporary disk space. Transbase breaks all info to be fulltext-indexed into single portions to be processed at at time. The performance increases with the size of the portions.

It is therefore recommended to specify in a CREATE FULLTEXT INDEX statement the capacity of the available disk space in the scratch directory. For example if it is assured that 60 MB will be available, then the statement might look like:

```
CREATE FULLTEXT INDEX x ON f(fb) SCRATCH 60 MB
```

Note, however, that the scratch area is shared among all applications on the database.

12.5.3 Tuple Deletion

Deletion of tuples from a table is slow if the table has at least one fulltext index. The deletion takes time which is proportional (linear) to the size of the fulltext index.

Note additionally, that it is much faster to delete several tuples in one single DELETE statement rather than to delete the tuples one at a time with several DELETE statements.

Chapter 13

The Transbase Data Dictionary

The Transbase data dictionary is a set of system tables which define the database structure.

Permissions on SystemTables: The data dictionary is owned by tbadmin (the database administrator), i.e. only tbadmin is allowed to directly update the data dictionary (discouraged). Other users, however, are allowed to update the data dictionary indirectly via DDL statement.

All users are allowed to read the data dictionary, i.e. to retrieve information about the database structure. Reading the data dictionary is in no way different from reading user tables.

Locks on SystemTables: For read access of system tables, a read lock is set as would also be set for user tables. However, to avoid bottlenecks on the system tables, Transbase automatically releases the read locks on system tables after the evaluation of the corresponding query. This means that repeated read accesses to the system tables might produce different results, namely if - in the meantime - a DDL transaction has been committed.

Note that this short lock duration does not mean that a reader sees an inconsistent catalog (since a read lock is held during the read operation). It also does not mean that serialization of transactions which access user tables is corrupted.

Summary of SystemTables: The data dictionary consists of the following tables:

sysuser

systable

syscolumn

sysindex

sysview
systablepriv
syscolumnpriv
sysviewdep
sysblob
sysconstraint
sysrefconstraint
sysdomain
loadinfo (CD databases only)

The following sections discuss each table in detail.

13.1 The sysuser Table

The sysuser table contains an entry for each registeured database user.

sysuser	
username	CHAR(*)
userclass	CHAR(*)
passwd	CHAR(*)
userid	INTEGER

Table 13.1: Structure of sysuser

Primary key: (username)

username of sysuser: any character string up to 30 characters.

userclass of sysuser: has one of the following values: "no access", "access", "resource", "dba"

Class "no access" cannot login on the database. If a user is revoked access privilege the userclass changes to 'no access'. Thus the user remains registered (and his objects if any are retained).

Class "access" is allowed to access database tables on which he has privileges. He is not allowed to create tables, views or indexes.

Class "resource" has privileges as in class "access" plus the privilege to create database objects, i.e. tables, views, and indexes.

Class "dba" has all access rights on the database, including the privilege to add or drop users, to drop objects where he is not owner , and to update the data dictionary tables manually. Strictly speaking, a user in class "dba" bypasses all privilege checks, i.e. no dba privileges are stored explicitly.

passwd of sysuser: Contains the encrypted password of the user. The encrypt algorithm is the same as used within the UNIX kernel. It uses only the first eight characters of the password string, i.e. only eight characters of a password are significant. Note that the user's password is not stored anywhere in unencrypted form.

userid of sysuser: Gives a unique identification for the user. This userid is used in other system tables to refer to users, e.g. in the systable table to denote the owner of a table.

Explanation: Upon creation of a database, two users are recorded in sysuser , namely "public" with userclass 'no access' and userid 0 and "tbadmin" with userclass "dba" and userid 1. Both users have assigned an empty string password (" ").

The user "public" is a dummy user which is used for implementation of the PUBLIC mechanism in the *GrantPrivilegeStatement*.

13.2 The systable Table

The systable table contains a single entry for each table or view defined in the database. An entry for each system table is given, too.

systable	
tname	CHAR(*)
owner	INTEGER
ttype	CHAR(*)
segno	INTEGER
colno	INTEGER
date	DATETIME[YY:MS]
cluster	SMALLINT

Table 13.2: Structure of systable

Primary key: (tname)

tname of systable: Identifier (see the TB/SQL reference manual) of at most 30 characters.

owner of systable: Denotes the owner of the table or view by the user's userid. To retrieve the owner 's username, join the tables sysuser and systable (see example below).

ttype of systable: The entry has one of the values "R","r","V","v","S". A user table has the value "R" or "r", where "R" is used for a table created WITH IKACCESS (this is the default) and "r" for a table created WITHOUT IKACCESS.

User views are denoted with entry "V".

System tables which are visible to the user have entry "v". In effect, all system tables described in this manual are views.

segno of systable: A positive integer value which denotes the physical segment number of the table or is a negative integer value if the entry is a view. segno is used in other system tables to identify a table or view uniquely.

colno of systable: Contains the number of the columns (fields) of the given table or view.

date of systable: Contains the creation time of the table or view (type tt DATETIME[YY:MS]).

cluster of systable: Contains the cluster number of the table as specified in the `CREATE TABLE` statement (0 is default). Only relevant for CD Databases.

Note: It is discouraged to use the segno of a user table as identifier in other user tables because the segno of a table might change when the database is archived and restored.

Example: Retrieve the table names and owners of non-system tables :

```
SELECT tname, username FROM systable , sysuser
WHERE owner  = userid  AND ttype <> 'S'
```

13.3 The syscolumn Table

The Syscolumn table contains a single entry for each field of each table or view defined in the database.

Primary key: (tsegno, cpos)

tsegno of syscolumn: Identifies the table or view to which the entry belongs. The name of the table can be retrieved via a join between systable and syscolumn on the fields tsegno and segno of syscolumn and systable , resp.

cname of syscolumn: Contains name of the column of the table or view (at most 30 characters).

syscolumn	
tsegno	INTEGER
cname	CHAR(*)
ctype	CHAR(*)
domainname	CHAR(*)
defaultvalue	CHAR(*)
¡not used¿	
cpos	INTEGER
ckey	INTEGER
notnull	CHAR(*)

Table 13.3: Structure of syscolumn

ctype of syscolumn: contains the base data type of the column. The data type is given as specified in the corresponding `CREATE TABLE` statement. Although data type specifications in TB/SQL are case-insensitive, strings stored in field ctype are all lower-case, namely: "bigint", "integer", "small-int", "tiny-int", "float", "double", "numeric(p,s)", "char(p)", "char(*)", "binchar(p)", "binchar(*)", "datetime[xx:zz]", "timespan[xx:zz]", "bool", "blob".

domainname of syscolumn: Contains the domainname if a domain has been used for field definition else NULL. Note that even if a domain has been used, its base type is recorded in field ctype.

defaultvalue of syscolumn: Contains the literal representation of the default value of the field. If no default value has been specified, the value NULL (again explicitly represented as literal) is stored.

cpos of syscolumn: Gives the position of the field in the table (first position is 1).

ckey of syscolumn: Indicates whether the corresponding column of a table belongs to the primary key (value ¿= 1) or not (value = 0).

If the key is not compound, all fields have ckey=0 except the primary-key field which has value 1. For a compound key, see example below.

notnull of syscolumn: If the `CREATE TABLE` statement has specified a `NOT NULL` clause, the field notnull has the value "Y", otherwise "N".

Example: The DDL statement

```
CREATE TABLE quotations (
Partno    INTEGER,
```

```
suppno    INTEGER,
price    NUMERIC(8,2) NOT NULL,
PRIMARY KEY (suppno, partno)
)
```

would produce the following three entries in syscolumn (not all fields are shown!):

cname	*ctype*	*cpos*	*ckey*	*notnull*
partno	integer	1	2	N
suppno	integer	2	1	N
price	numeric(8,2)	3	0	Y

13.4 The sysindex Table

For each index defined for the database, entries in the sysindex table are made. If an n-field index is defined, n entries in sysindex are used to describe the index structure.

sysindex	
tsegno	INTEGER
iname	CHAR(*)
weight	INTEGER
cpos	INTEGER
isuniq	CHAR(*)
isegno	INTEGER
wlistsegno	INTEGER
stowosegno	INTEGER
charmapsegno	INTEGER
delimsegno	INTEGER
ftxttype	CHAR(*)
wlisttype	CHAR(*)

Table 13.4: Structure of sysindex

Primary key: (tsegno, iname, weight)

tsegno of sysindex: Identifies the base table which the index refers to. To retrieve the name of the table, perform a join between systable and sysindex on fields tsegno,segno.

iname of sysindex: Stores the name of the index (Identifier of at most 30 characters). Index names are unique with respect to the database.

weight of sysindex: Serial number for the fields of one index, starting at 1 with the first field specified in the *CreateIndexStatement*.

cpos of sysindex: Identifies a field of the base table which the index refers to. To retrieve the name of the field, perform a join between syscolumn and sysindex on the fields (tsegno, cpos) in each table (see the example below).

isuniq of sysindex: Contains string "Y" if the index is specified as "UNIQUE" , "N" otherwise.

isegno of sysindex: The field isegno contains the physical segment number of the index. Note that indexes are stored as B*-trees in physical segments.

wlistsegno of sysindex: Contains for a fulltext index the segment number of the wordlist table, else NULL.

stowosegno, charmapsegno, delimsegno of sysindex: Contain NULL value for a non-fulltext index.

Contain segment number for the stopword table, charmap table, delimiters table, resp., if they have been defined else NULL.

ftxttype of sysindex: Contains NULL value for a non-fulltext index else one of values "positional" or "word". "positional" is for a POSITIONAL fulltext index, "word" is the default.

wlisttype of sysindex: Contains NULL value for a non-fulltext index.

Contains value "fix" for a specified wordlist, "var" is the default.

Example: The DDL statement

```
CREATE INDEX partprice ON quotations (partno, price)
```

would produce the following two entries in sysindex (only some fields of sysindex are shown):

iname	...	*weight*	*cpos*	*isuniq*
partprice	...	1	1	N
partprice	...	2	3	N

Example: To retrieve the names of the index fields in proper order, the following statement is used:

```
SELECT t.tname, c.cname, i.iname, i.weight
FROM systable t, syscolumn  c,
sysindex  i
```

```
WHERE t.segno=c.tsegno
AND c.tsegno=i.tsegno AND c.cpos=i.cpos
AND i.iname='partprice'
ORDER BY i.weight
```

13.5 The sysview Table

Contains one tuple for each view defined in the database.

sysview	
vsegno	INTEGER
viewtext	CHAR(*)
checkopt	CHAR(*)
updatable	CHAR(*)
viewtree	CHAR(*)

Table 13.5: Structure of sysview

Primary key: (vsegno)

vsegno of sysview: Contains (negative) integer value which uniquely identifies the view. The same value is used e.g. in systable and syscolumn to refer to the view.

viewtext of sysview: Contains SQL SELECT statement which defines the view. This may serve as a reminder for the user and also is is used by Transbase whenever the view is used in a SQL statement.

checkopt of sysview: Contains "Y" if the view has been defined WITH CHECK OPTION else "N".

updatable of sysview : Contains "Y" if the defined view is updatable else "N".

If a view is not updatable , only SELECT statements may be applied to the view. If a view is updatable , UPDATE, INSERT, and DELETE statements may be applied, too.

For the definition of updatability and for the semantics of updates on views see the TB/SQL Reference Manual.

viewtree of sysview: Reserved for internal use of Transbase.

Note: Additional information for a view is contained in systable and syscolumn like for base tables (join systable, syscolumn, sysview).

13.6 The sysviewdep Table

Contains dependency information of views and base tables.

A view may be defined on base tables as well as on other previously defined views. If a base table or view is dropped, all views depending on this base table or view are dropped automatically. For this reason the dependency information is stored in sysviewdep.

sysviewdep	
basesegno	INTEGER
vsegno	INTEGER

Table 13.6: Structure of sysviewdep

Primary key: (basesegno, vsegno)

basesegno of sysviewdep: Contains the segment number of the base table or view on which the view identified by vsegno depends. basesegno is positive or negative depending on being a base table or view.

vsegno of sysviewdep: Identifies the view which depends on the base table or view identified by basesegno. vsegno always is negative.

13.7 The sysblob Table

Contains the information about all BLOB container segments. For each user base table which has at least one column of type BLOB there is one tuple in the sysblob table. All BLOB objects of the table are stored in the denoted BLOB container. Note that only base tables but not views which contain BLOB columns contribute to sysblob.

sysblob	
rsno	INTEGER
bcont	INTEGER

Table 13.7: Structure of sysblob

Primary key: (rsno)

rsno of sysblob: Contains segment number of the base table containing BLOB field(s).

bcont of sysblob: Contains segment number of the BLOB container of the base table.

13.8 The systablepriv Table

Describes the privileges applying to tables of the database. Note that UPDATE privileges can be specified columnwise; those privileges are defined in table syscolumnpriv.

systablepriv contains for each table all users who have privileges on this table. Furthermore it contains who granted the privilege and what kind of privilege it is.

systablepriv	
grantee	INTEGER
tsegno	INTEGER
grantor	INTEGER
sel_priv	CHAR(*)
del_priv	CHAR(*)
ins_priv	CHAR(*)
upd_priv	CHAR(*)

Table 13.8: Structure of systablepriv

Primary key: (grantee, tsegno, grantor)

grantee,tsegno,grantor of systablepriv: These three fields describe who (i.e. the grantor) has granted a privilege to whom (i.e. the grantee) on which base table or view (tsegno). The kind of privilege(s) is described in the other fields.

grantor and grantee refer to field userid of table sysuser. tsegno refers to field tsegno of table systable.

sel_priv, del_priv, ins_priv, upd_priv of systablepriv: These fields describe privileges for SELECT, DELETE, INSERT, UDATE.

Each contains one of the values "N", "Y", "G", or in case of *updpriv* also "S".

"Y" means that grantee has received from grantor the corresponding privilege for the table or view identified by tsegno.

"G" includes "Y" and additionally the right to grant the privilege to other users, too.

"S" (only in *updpriv*) stands for "selective UPDATE privilege" ; it indicates that there exist entries in table syscolumnpriv which describe column-specific UPDATE privileges granted from grantor to grantee on table tsegno.

"N" is the absence of any of the above described privileges.

For each tuple, at least one of the fields selpriv, delpriv, inspriv, updpriv is different from "N".

Default: The owner of a table always has all privileges with GRANT OPTION on the table, by default. Those privileges are recorded in systablepriv, too. Namely, if user 34 creates a table identified by 73, the following entry in systablepriv is made:

(34, 73, 34, 'G', 'G', 'G', 'G')

13.9 The syscolumnpriv Table

The table syscolumnpriv describes the UPDATE privileges on columns of the database.

syscolumnpriv	
grantee	INTEGER
tsegno	INTEGER
grantor	INTEGER
cpos	INTEGER
upd_priv	CHAR(*)

Table 13.9: Structure of syscolumnpriv

Primary key: (grantee, tsegno, grantor, cpos)

grantee, tsegno, grantor, cpos of syscolumnpriv: These four fields describe who (*grantor*) has granted a privilege to whom (*grantee*) on which field (*cpos*) on which base table or view (*tsegno*). The kind of privilege(s) is described in the other fields.

grantor and *grantee* refer to field *userid* of table sysuser. *tsegno* refers to field *tsegno* of table systable.

upd_priv of syscolumnpriv: Contains one of the strings "Y" or "G".

"Y" means that *grantee* has received from *grantor* the UPDATE privilege for the specified field on the specified table or view.

"G" includes "Y" and additionally the right to grant the privilege to other users, too.

Note: If a user grantee has been granted the same update privilege ("Y" or "G") on all fields on tsegno from the same grantor, then these privileges are recorded via a corresponding single tuple in table systablepriv.

13.10 The sysdomain Table

Contains at least one tuple for each created DOMAIN of the database. Several tuples for one domain are present if more than one check constraints are defined on the domain.

sysdomain	
name	CHAR(*)
owner	INTEGER
basetype	CHAR(*)
defaultvalue	CHAR(*)
constraintname	CHAR(*)
attributes	CHAR(*)
constrainttext	CHAR(*)

Table 13.10: Structure of sysdomain

Primary key: (name,constraintname)

name of sysdomain: Contains the name of the domain.

owner of sysdomain: Contains the userid of the creator of the domain.

basetype of sysdomain: Contains the basetype of the domain.

defaultvalue of sysdomain: Contains the defaultvalue (in literal representation) of the domain if there has been declared one otherwise the literal NULL.

constraintname, attributes, constrainttext of sysdomain: These fields describes a domain constraint if there has been defined one else they are all NULL. Field attributes contains the value "IMMEDITATE" (for use in later versions), constraintname stores the user defined constraintname if any else NULL, constrainttext stores the search condition of the constraint.

Note: If `n > 1` constraints are defined, all n tuples redundantly have the same owner, basetype, defaultvalue.

13.11 The sysconstraint Table

Contains one tuple for each table constraint. Constraint types are PRIMARY KEY or CHECK(...) constraints.

Primary key: (segno,constraintname)

sysconstraint	
segno	INTEGER
constraintname	CHAR(*)
attributes	CHAR(*)
constrainttext	CHAR(*)
cpos	INTEGER

Table 13.11: Structure of sysconstraint

segno of sysconstraint: Contains the segment of the table the constraint refers to.

constraintname, attributes, constrainttext, cpos: These fields describe one constraint: constraintname stores the user defined constraintname if any else a unique system defined constraintname. Field attributes has the constant value "IMMEDIATE" (for more general use in later versions), constrainttext stores the constraint text which is either of the form PRIMARY KEY (...) or CHECK(...). Field cpos has value -1 except in the case that the constraint (let's say cd) has implicitly been created for the table by the fact that a domain dom has been DROPped with CASCADE by a user and dom has been used for a field f by the table and dom had the constraint cd. In this case, the constrainttext is the same as it was for the domain constraint (constains keyword VALUE) and field cpos has value f.

Note: Reference constraints (FOREIGN KEY ...) are stored in table sysrefconstraint.

13.12 The sysrefconstraint Table

Contains tuples to describe reference constraints (FOREIGN KEY ...).

sysrefconstraint	
constraintname	CHAR(*)
attributes	CHAR(*)
srcsegno	INTEGER
srccpos	INTEGER
tarsegno	INTEGER
tarcpos	INTEGER
delact	CHAR(*)
updact	CHAR(*)

Table 13.12: Structure of sysrefconstraint

Primary key: (srcsegno,constraintname)

constraintname of sysrefconstraint: Contains the name of the constraint (explicitly by user or system defined).

attributes of sysrefconstraint: Contains value "IMMEDIATE" (for more general use in later versions).

srcsegno, tarsegno of sysrefconstraint: Contain the segment number of the referencing table and the referenced table, resp.

delact of sysrefconstraint: Contains the action to be performed with a referencing tuple rf when rf looses its referenced tuple rd due to a deletion of rd.

The field has one of the string values "NO ACTION" , "CASCADE" , "SET NULL" , "SET DEFAULT". For the semanticsof these actions, (see TB/SQL Reference Manual, *CreateTableStatement*, *ForeignKey*).

updact of sysrefconstraint: Contains the action to be performed on a referencing tuples rf when rf looses its referenced tuple rd due to a update of rd. The field has the constant string values "NO ACTION" and is for more general use in later versions. For the semantics of this action, (see TB/SQL Reference Manual, *CreateTableStatement*, *ForeignKey*).

srccpos, tarcpos of sysrefconstraint: Contains a pair of field positions (¿=1) which correspond to the referencing and referenced field of the reference constraint. If the reference constraint is defined over a n-ary field combination, then n tuples are in sysrefconstraint where all values except srcpos and tarcpos are identical and srcpos and tarcpos have the values of the corresponding field positions.

Example:

```
CREATE  TABLE  sampletable
(   keysample INTEGER,
    f1        INTEGER,
    f2        INTEGER,
    CONSTRAINT ctref
        FOREIGN KEY (f1,f2)
        REFERENCES reftable(key1,key2)
        ON DELETE CASCADE
)
```

Assume that sampletable and reftable have numbers 13 and 19, resp., and that reftable has fields key1 and key2 on positions 7 and 8.

Then the following two tuples are inserted in sysrefconstraint:

constraintname	*attributes*	*srcsegno*	*srccpos*	*tarsegno*	*tarcpos*	*delact*	*updact*
ctref	IMMEDIATE	13	2	19	7	CASCADE	NO ACTION
ctref	IMMEDIATE	13	3	19	8	CASCADE	NO ACTION

13.13 The loadinfo Table

Describes the status of the diskcache in CD Retrieval Databases. It is not present in Transbase Standard Databases. It is implemented as view and thus also recorded in the sysview table.

For each page which is on the diskcache there is one tuple in loadinfo.

loadinfo	
segment	INTEGER
rompage	INTEGER
diskpage	INTEGER
flag	INTEGER

Table 13.13: Structure of loadinfo

Primary key: (segment,rompage)

segment of loadinfo: Contains the segment number of the page.

rompage of loadinfo: Contains the page number in the CD-ROM address space of the page.

diskpage of loadinfo: Contains the page number in the diskfile address space of the page.

flag of loadinfo: Contains the value 1 if the page has been fetched via a LOAD STATEMENT and the value 2 if the page has been stored due to a INSERT or UPDATE or DELETE STATEMENT.

Chapter 14

Precedence of Operators

Table 14.1 below defines the precedence of operators. A precedence of 1 means highest precedence. Associativity within a precedence level is from left to right.

Precedence	*Operators*
1	SUBRANGE, SIZE OF
2	<timeselector> OF, CONSTRUCT
3	CAST
4	PRIOR, +, - (unary) , BITNOT
5	BITAND, BITOR
6	*, /
7	+, - (binary), ||, < <= = <> >= >
8	LIKE, MATCHES, IN, BETWEEN, SUBSET, CONTAINS
9	NOT
10	AND
11	OR
12	SELECT
13	INTERSECT
14	UNION, DIFF

Table 14.1: Precedence of operators

Appendix A

Transbase SQL Keywords

ACCESS
ACTION
ADD
ALL
ALTER
AND
ANY
AS
ASC
AVG
BASENAME
BETWEEN
BIGINT
BINARY
BINCHAR
BITAND
BITNOT
BITOR
BITS
BITS2
BLOB
BLOBACCESS
BOTH
BOOL
BY
CACHE
CALL
CASCADE
CASE
CAST
CHAR
CHARACTER
CHARACTER_LENGTH
CHECK
CLOSURE
COALESCE
COLUMN
CONSTRAINT
CONSTRUCT
CONTAINS
CORRESPONDING
COUNT
COUNTBIT
CREATE
CROSS
CURRENT
CURRENTDATE
CURRVAL
DATE
DATETIME
DBA
DD
DECIMAL
DECODE
DEFAULT
DELETE
DELIM
DESC
DIFF
DIRNAME
DISK
DISTINCT
DOMAIN
DOUBLE
DROP
ELSE
END
ESCAPE
EXCLUSIVE
EXISTS
EXTERNAL
FALSE
FI
FILEREF
FILL
FINDBIT
FLOAT
FOR
FOREIGN
FROM
FULL
FULLTEXT
FUNCTION
GENTREE
GRANT
GROUP
HAVING

HH
IF
IKACCESS
IN
INCLUSIVE
INDEX
INNER
INPOLYGON
INSENSITIVE
INSERT
INSTR
INTEGER
INTERSECT
INTERVAL
INTO
IS
JOIN
KEY
LEADING
LEAF
LEFT
LEVELS
LIKE
LISP
LOAD
LOCAL
LOCK
LOWER
LTRIM
MAKEBIT
MATCHES
MAX
MI
MIN
MO
MONEY
MS
NATURAL
NEXTVAL
NOT
NULL
NULLEQUAL
NULLIF
NUMERIC
NVL
OF
OFF
ON
OPTION
OR
ORDER
OUTER
PASSWORD
POSITION
POSITIONAL
PRIMARY
PRIVILEGES
PROCEDURE
PUBLIC
READ
REAL
REFERENCES
RELATION
REPLACE
REPLICATE
RESOURCE
RESTRICT
RESULTCOUNT
REVOKE
RIGHT
RTRIM
SELECT
SENSITIVE
SEQUENCE
SET
SIGN
SIZE
SMALLINT
SOME
SORTED
SPOOL
SS
STRING
SUBRANGE
SUBSET
SUBSTR
SUBSTRING
SUM
SURROGATE
SYSDATE
SYSKEY
TAB
TABLE
TBMODE
THEN
TIME
TIMESPAN
TIMESTAMP
TINYINT
TO
TO_CHAR
TRAILING
TRIGGER
TRIM
TRUE
TRUNC
TYPE
UBVALUEFUNC
UNGROUP
UNION
UNIQUE
UNLOAD
UNLOCK
UPDATE
UPPER
USAGE
USER
USING
VALUES
VARBINARY
VARCHAR
VARYING
VIEW
WEEKDAY
WHEN
WHERE
WITH
WITHOUT
YY

Appendix B

Database Schema SAMPLE

The database 'SAMPLE' used in the exercises throughout this manual consists of three tables named SUPPLIERS, PARTS, and INVENTORY the structure and contents of which is given in the following tables B.1, B.2 and B.3 below.

suppno	name	address
51	DEFECTO PARTS	16 BUM ST., BROKEN HAND WY
52	VESUVIUS, INC.	512 ANCIENT BLVD., POMPEII NY
53	ATLANTIS CO.	8 OCEAN AVE., WASHINGTON DC
54	TITANIC PARTS	32 LARGE ST., BIG TOWN TX
57	EAGLE HARDWARE	64 TRANQUILITY PLACE, APOLLO MN
61	SKY PARTS	128 ORBIT BLVD., SIDNEY
64	KNIGHT LTD.	256 ARTHUR COURT, CAMELOT

Table B.1: Table SUPPLIERS

partno	description	qonhand
207	GEAR	75
209	CAM	50
221	BOLT	650
222	BOLT	1250
231	NUT	700
232	NUT	1100
241	WASHER	6000
285	WHEEL	350
295	BELT	85

Table B.2: Table PARTS

suppno	partno	price	delivery_time	qonorder
50	221	.30	10	50
51	221	.30	10	50
51	231	0.10	10	0
53	222	0.25	15	0
53	232	0.10	15	200
53	241	0.08	15	0
54	209	18.00	21	0
54	221	0.10	30	150
54	231	0.04	30	200
54	241	0.02	30	200
57	285	21.00	4	0
57	295	8.50	21	24
61	221	0.20	21	0
61	222	0.20	21	200
61	241	0.05	21	0
64	207	29.00	14	20
64	209	19.50	7	7

Table B.3: Table INVENTORY